

**27**  
**SERIA**  
**PC**



**LIMBAJELE  
C SI C++  
PENTRU  
INCEPATORI**  
VOLUMUL I

**LIMBAJUL C**



LIMBAJELE  
C SI C++  
PENTRU  
INCEPATORI  
LIMBAJUL C  
REEDITARE  
VOLUMUL I

CLUJ-NAPOCA 1995

Autor

**Liviu Negrescu**

A confruntat cu originalul  
Irina Mitrov

Editura Albastră



Coordonator serie  
Smaranda Derveșteanu

Tehnoredactare computerizată  
Codruța Poenaru

Coperta  
Liviu Derveșteanu

Tipărit

Comanda nr. 123  
Imprimeria Ardealul Cluj



Tiraj 2000 exemplare

# CUPRINS

PREFAȚĂ . . . . .	13
-------------------	----

INTRODUCERE . . . . .	19
-----------------------	----

1. NOȚIUNI DE BAZĂ . . . . .	33
------------------------------	----

1.1. Nume . . . . .	33
1.2. Cuvinte cheie . . . . .	34
1.3. Tipuri de bază . . . . .	34
1.4. Structura unei funcții . . . . .	37
1.5. Comentariu . . . . .	40
1.6. Constante . . . . .	41
1.6.1. Constante întregi . . . . .	41
1.6.2. Constante flotante . . . . .	44
1.6.3. Constantă caracter . . . . .	45
1.6.4. Șir de caractere . . . . .	48
1.7. Caractere sau spații albe (white spaces) . . . . .	50
1.8. Variabile simple, tablouri și structuri . . . . .	51
1.9. Declarația de variabilă simplă . . . . .	53
1.10. Declarația de tablou . . . . .	54
1.11. Apelul și prototipul funcțiilor . . . . .	56
1.12. Procesare . . . . .	60
1.12.1. Incluseri de fișiere cu texte sursă . . . . .	60
1.12.2. Substituiri de succesiuni de caractere . . . . .	62

2. INTRĂRI/IEȘIRI STANDARD . . . . .	65
--------------------------------------	----

2.1. Funcțiile getch și getche . . . . .	65
2.2. Funcția putchar . . . . .	66
Exerciții . . . . .	67
2.3. Macrourile getchar și putchar . . . . .	69
Exerciții . . . . .	70
2.4. Funcțiile gets și puts . . . . .	71
Exerciții . . . . .	72
2.5. Funcția printf . . . . .	74
2.5.1. Litera c . . . . .	77



2.5.2.	Litera s . . . . .	77
2.5.3.	Litera d . . . . .	78
2.5.4.	Litera o . . . . .	79
2.5.5.	Literele x și X . . . . .	80
2.5.6.	Litera u . . . . .	80
2.5.7.	Litera l . . . . .	80
2.5.8.	Litera f . . . . .	80
2.5.9.	Literele e și E . . . . .	81
2.5.10.	Literele g și G . . . . .	82
2.5.11.	Litera L . . . . .	82
	Exerciții . . . . .	82
2.6.	Funcția scanf . . . . .	88
2.6.1.	Litera c . . . . .	90
2.6.2.	Litera s . . . . .	91
2.6.3.	Litera d . . . . .	93
2.6.4.	Litera o . . . . .	95
2.6.5.	Literele x sau X . . . . .	95
2.6.6.	Litera u . . . . .	95
2.6.7.	Litera f . . . . .	96
2.6.8.	Litera l . . . . .	96
2.6.9.	Litera L . . . . .	96
	Exerciții . . . . .	96

### 3. EXPRESII . . . . . 101

3.1.	Operand . . . . .	101
3.2.	Operatori . . . . .	102
3.2.1.	Operatori aritmetici . . . . .	103
	Exerciții . . . . .	104
3.2.2.	Regula conversiilor implicite . . . . .	110
	Exerciții . . . . .	111
3.2.3.	Operatorii de relație . . . . .	113
	Exerciții . . . . .	114
3.2.4.	Operatorul de egalitate . . . . .	115
3.2.5.	Operatori logici . . . . .	116
	Exerciții . . . . .	118
3.2.6.	Operatori logici pe biți . . . . .	121
3.2.7.	Operatorul de atribuire . . . . .	126
	Exerciții . . . . .	129
3.2.8.	Operatorii de incrementare și decrementare . . . . .	131
3.2.9.	Operatorul de forțare a tipului sau conversie explicită . . . . .	132
	Exerciții . . . . .	132

3.2.10.	Operatorul dimensiune	134
3.2.11.	Operatorul adresă	135
3.2.12.	Operatorii paranteză	136
3.2.13.	Operatorii condiționali	136
	Exerciții	138
3.2.14.	Operatorul virgulă	140
	Exerciții	140
3.2.15.	Alți operatori ai limbajului C	141
3.2.16.	Tabela cu prioritățile operatorilor limbajului C	142

#### 4. INSTRUȚIUNI . . . . . 143

4.1.	Instrucțiunea vidă	144
4.2.	Instrucțiunea expresie	145
	Exerciții	146
4.3.	Instrucțiunea expresie	147
4.4.	Instrucțiunea compusă	148
	Exerciții	149
4.5.	Funcția standard exit	157
	Exerciții	158
4.6.	Instrucțiunea while	160
	Exerciții	160
4.7.	Instrucțiunea for	181
	Exerciții	184
4.8.	Instrucțiunea do-while	189
	Exerciții	190
4.9.	Instrucțiunea continue	195
4.10.	Funcțiile standard scanf și printf	195
	Exerciții	197
4.11.	Instrucțiunea break	199
	Exerciții	199
4.12.	Instrucțiunea switch	203
	Exerciții	205
4.13.	Instrucțiunea goto	208
4.14.	Programarea procedurală, funcții, apelul și revenirea din ele	210
	Exerciții	215
4.15.	Apel prin valoare și apel prin referință	224
	Exerciții	225

<b>5.</b>	<b>CLASE DE MEMORIE</b>	<b>247</b>
5.1.	Variabile globale	247
5.2.	Variabile locale	249
5.3.	Alocarea parametrilor	254
5.4.	Utilizarea parametrilor și a variabilelor globale	254
5.5.	Variabile registru	255
	Exerciții	256
<b>6.</b>	<b>INIȚIALIZARE</b>	<b>264</b>
6.1.	Inițializarea variabilelor simple	264
	Exerciții	266
6.2.	Inițializarea tablourilor	267
	Exerciții	272
<b>7.</b>	<b>PROGRAMARE MODULARĂ</b>	<b>282</b>
	Exerciții	283
<b>8.</b>	<b>POINTERI</b>	<b>294</b>
8.1.	Declarația de pointer și tipul pointer	297
8.2.	Realizarea apelului prin referință	301
	Exerciții	302
8.3.	Legătura dintre pointeri și tablouri	312
8.4.	Operații cu pointeri	314
8.4.1.	Operații de incrementare și decrementare a pointerilor	314
8.4.2.	Adunarea și scăderea unui întreg dintr-un pointer	315
8.4.3.	Compararea a doi pointeri	316
8.4.4.	Diferența a doi pointeri	318
	Exerciții	318
8.5.	Modificatorul const	322
8.6.	Funcții standard utilizate la prelucrarea șirurilor de caractere	326
8.6.1.	Lungimea unui șir de caractere	327
8.6.2.	Copiarea unui șir de caractere	328
8.6.3.	Concatenarea șirurilor de caractere	330
	Exerciții	333
8.7.	Expresie lvalue	340

8.8.	Alocarea dinamică a memoriei . . . . .	342
	Exerciții . . . . .	345
8.9.	Utilizarea tablourilor de pointeri la prelucrări de date de tip șir de caractere . . . . .	350
	Exerciții . . . . .	351
8.10.	Tratarea parametrilor din linia de comandă . . . . .	355
	Exerciții . . . . .	357
8.11.	Pointeri spre funcții . . . . .	359
	Exerciții . . . . .	361
<b>9.</b>	<b>RECURSIVITATE . . . . .</b>	<b>366</b>
	Exerciții . . . . .	370
<b>10.</b>	<b>STRUCTURI ȘI TIPURI DEFINITE DE UTILIZATOR . . . . .</b>	<b>377</b>
10.1.	Declarația de structură . . . . .	378
10.2.	Accesul la elementele unei structuri . . . . .	383
10.3.	Asignări de nume pentru tipuri de date . . . . .	387
	Exerciții . . . . .	391
10.4.	Reuniune . . . . .	429
	Exerciții . . . . .	434
10.5.	Cîmp . . . . .	445
	Exerciții . . . . .	447
10.6.	Tipul enumerare . . . . .	462
10.7.	Date definite recursiv . . . . .	465
	Exerciții . . . . .	468
<b>11.</b>	<b>LISTE . . . . .</b>	<b>478</b>
11.1.	Lista simplu înlănțuită . . . . .	479
11.1.1.	Crearea unei liste simplu înlănțuite . . . . .	480
11.1.2.	Accesul la un nod al unei liste simplu înlănțuite . . . . .	484
11.1.3.	Inserarea unui nod într-o listă simplu înlănțuită . . . . .	487
11.1.3.1.	Inserarea unui nod într-o listă simplu înlănțuită înaintea primului ei nod . . . . .	487
11.1.3.2.	Inserarea unui nod într-o listă simplu înlănțuită înaintea unui nod precizat printr-o cheie . . . . .	489
11.1.3.3.	Inserarea unui nod într-o listă simplu înlănțuită după un nod precizat printr-o cheie . . . . .	492

11.1.3.4.	Adăugarea unui nod la o listă simplu înlănțuită . . . . .	493
11.1.4.	Ștergerea unui nod dintr-o listă simplu înlănțuită . . . . .	495
11.1.4.1.	Ștergerea primului nod al unei liste simplu înlănțuite . . . . .	495
11.1.4.2.	Ștergerea unui nod precizat printr-o cheie, dintr-o listă simplu înlănțuită . . . . .	496
11.1.4.3.	Ștergerea ultimului nod dintr-o listă simplu înlănțuită . . . . .	498
11.1.5.	Ștergerea unei liste simplu înlănțuite . . . . .	499
	Exerciții . . . . .	500
11.2.	Stive și cozi . . . . .	508
	Exerciții . . . . .	513
11.3.	Listă circulară simplu înlănțuită . . . . .	522
11.3.1.	Crearea unei liste circulare . . . . .	524
11.3.2.	Accesul la un nod al unei liste circulare . . . . .	526
11.3.3.	Inserarea unui nod într-o listă circulară . . . . .	527
11.3.3.1.	Inserarea unui nod înaintea unuia precizat printr-o cheie de tip int . . . . .	528
11.3.3.2.	Inserarea unui nod după un nod precizat printr-o cheie de tip int . . . . .	529
11.3.4.	Ștergerea unui nod dintr-o listă circulară . . . . .	530
11.3.5.	Ștergerea unei liste circulare . . . . .	531
	Exerciții . . . . .	531
11.4.	Listă dublu înlănțuită . . . . .	537
11.4.1.	Crearea unei liste dublu înlănțuite . . . . .	539
11.4.2.	Inserarea unui nod într-o listă dublu înlănțuită . . . . .	540
11.4.2.1.	Inserarea unui nod într-o listă dublu înlănțuită înaintea primului ei nod . . . . .	540
11.4.2.2.	Inserarea unui nod într-o listă dublu înlănțuită înaintea unui nod precizat printr-o cheie . . . . .	541
11.4.2.3.	Inserarea unui nod într-o listă dublu înlănțuită după unul precizat printr-o cheie . . . . .	542
11.4.2.4.	Inserarea unui nod într-o listă dublu înlănțuită după ultimul nod (adăugarea unui nod la o listă dublu înlănțuită) . . . . .	543
11.4.3.	Ștergerea unui nod dintr-o listă dublu înlănțuită . . . . .	544
11.4.3.1.	Ștergerea primului nod al unei liste dublu înlănțuite . . . . .	545
11.4.3.2.	Ștergerea unui nod dintr-o listă dublu înlănțuită precizat printr-o cheie . . . . .	545
11.4.3.3.	Ștergerea ultimului nod al unei liste dublu înlănțuite . . . . .	546
	Exerciții . . . . .	548

<b>12.</b>	<b>ARBORI</b>	<b>554</b>
12.1.	Inserarea unui nod frunză într-un arbore binar	564
12.2.	Accesul la un nod al unui arbore binar	567
12.3.	Parcursarea unui arbore binar	568
12.3.1.	Parcursarea arborilor binari în preordine	575
12.3.2.	Parcursarea arborilor binari în înordine	576
12.3.3.	Parcursarea arborilor binari în postordine	576
12.4.	Ștergerea unui arbore binar	577
	Exerciții	577
<b>13.</b>	<b>TABELE</b>	<b>589</b>
13.1.	Tabela de cuvinte rezervate	590
13.2.	Tabela de dispersie	592
	Exerciții	599
<b>14.</b>	<b>SORTARE</b>	<b>609</b>
14.1.	Sortare Shell	610
14.2.	Sortare rapidă	615
	Exerciții	633
<b>15.</b>	<b>DIN NOU DESPRE PREPROCESARE ÎN C</b>	<b>640</b>
15.1.	Definiții și apeluri de macroui	640
15.2.	Compilare condiționată	646
	Exerciții	649
<b>16.</b>	<b>INTRĂRI/IEȘIRI</b>	<b>653</b>
16.1.	Nivelul inferior de prelucrare a fișierelor	655
16.1.1.	Deschiderea unui fișier	655
16.1.2.	Citirea dintr-un fișier	658
16.1.3.	Scrierea într-un fișier	659
16.1.4.	Poziționarea într-un fișier	659
16.1.5.	Închiderea unui fișier	660
	Exerciții	661
16.2.	Nivelul superior de prelucrare a fișierelor	667
16.2.1.	Deschiderea unui fișier	667
16.2.2.	Prelucrarea pe caracter a unui fișier	668



	Exerciții . . . . .	669
16.2.3.	Închiderea unui fișier . . . . .	670
	Exerciții . . . . .	670
16.2.4.	Intrări/ieșiri de șiruri de caractere . . . . .	671
16.2.5.	Intrări/ieșiri cu format . . . . .	672
16.2.6.	Vidarea zonei tampon a unui fișier . . . . .	673
16.2.7.	Poziționarea într-un fișier . . . . .	674
16.2.8.	Prelucrarea fișierelor binare . . . . .	675
16.3.	Ștergerea unui fișier . . . . .	676
16.4.	Redirecționarea fișierului de intrare/ieșire standard . . . . .	676
	Exerciții . . . . .	677

## **17. FUNCȚII STANDARD . . . . . 682**

17.1.	Macroui de clasificare . . . . .	684
17.2.	Macroui de transformare a caracterelor . . . . .	685
17.3.	Funcții de conversie . . . . .	685
17.4.	Funcții de calcul . . . . .	687
17.5.	Funcții pentru gestiunea memoriei . . . . .	689
17.6.	Funcții de control ale proceselor . . . . .	689
17.7.	Funcții pentru gestiunea datei și a orei . . . . .	690
17.8.	Diferite funcții de uz general . . . . .	691
17.9.	Tratarea erorilor . . . . .	693
	Exerciții . . . . .	694

## **18. GESTIUNEA ECRANULUI ÎN MOD TEXT . . . 702**

18.1.	Setarea ecranului în mod text . . . . .	704
18.2.	Definirea unei ferestre . . . . .	705
18.3.	Ștergerea unei ferestre . . . . .	705
18.4.	Gestiunea cursorului . . . . .	705
18.5.	Determinarea parametrilor ecranului . . . . .	707
18.6.	Modurile video alb/negru . . . . .	707
18.7.	Setarea culorilor . . . . .	708
	Exerciții . . . . .	708
18.8.	Gestiunea textelor . . . . .	710
	Exerciții . . . . .	712

<b>19.</b>	<b>GESTIUNEA ECRANULUI ÎN MOD GRAFIC . . . . .</b>	<b>729</b>
19.1.	Setarea ecranului în mod grafic . . . . .	729
	Exerciții . . . . .	733
19.2.	Gestiunea culorilor . . . . .	734
	Exerciții . . . . .	739
19.3.	Starea ecranului . . . . .	739
	Exerciții . . . . .	740
19.4.	Gestiunea textelor . . . . .	741
	Exerciții . . . . .	744
19.5.	Gestiunea imaginilor . . . . .	747
	Exerciții . . . . .	752
19.6.	Tratarea erorilor . . . . .	758
	Exerciții . . . . .	758
19.7.	Desenare și colorare . . . . .	763
	Exerciții . . . . .	770
	<b>BIBLIOGRAFIE . . . . .</b>	<b>781</b>

# PREFAȚĂ

Deceniul anilor '90 este deceniul consacrat *programării orientate spre obiecte* (în engleză Object Oriented Programming sau prescurtat OOP) la fel cum anii '70 au fost numiți anii *programării structurate*.

Programarea orientată spre obiecte este un *stil* de programare care permite abordarea eficientă a aplicațiilor complexe. Acesta se asigură pe baza:

- elaborării de componente reutilizabile, extensibile și ușor modificabile, fără a reprograma totul de la început;
- construirii de biblioteci de module extensibile,
- implementării simple a interacțiunilor programelor cu mediul de calcul prin utilizarea unor elemente standardizate.

Pe scurt, putem afirma că, programarea orientată spre obiecte permite să se implementeze sisteme complexe cu ajutorul unor componente individuale, standardizate, reutilizabile, extensibile și ușor modificabile.

Această concepție a influențat dezvoltarea limbajelor de programare. Ea s-a materializat începând din anii '60 când s-a introdus conceptul de *obiect* în limbajul *Simula*.

În principiu, un obiect are o structură de date bine precizată și totodată sînt definite operațiile care pot utiliza componentele lui.

*Simula* se consideră că se află la baza limbajelor care permit programarea orientată spre obiecte.

În cele ce urmează vom spune despre un limbaj de programare că este *orientat spre obiecte* dacă permite programarea orientată spre obiecte.

În deceniul anilor '70, programarea orientată spre obiecte s-a utilizat mai ales prin intermediul limbajului *Smalltalk*. Așa cum s-a amintit mai sus, în acest deceniu s-au pus bazele programării structurate care a avut un succes deosebit prin utilizarea limbajului *Pascal*.

În anul 1972 a apărut *limbajul C*. Acesta este un limbaj de programare care are o destinație universală. Autorii acestui limbaj sînt Dennis M. Ritchie și Brian W. Kernighan de la Bell Laboratories.

Limbajul *C* a fost proiectat în ideea de a asigura implementarea portabilă a sistemului de operare UNIX. Un rezultat direct al acestui fapt este acela că programele scrise în limbajul *C* au o *portabilitate* foarte bună.

În mod intuitiv, spunem că un program este portabil dacă el poate fi

transferat ușor de la un tip de calculator la altul. În principiu, toate limbajele de nivel înalt asigură o anumită portabilitate a programelor. La ora actuală se afirmă că programele scrise în C sînt cele mai portabile.

Prin anii '80 interesul pentru programarea orientată spre obiecte a crescut, ceea ce a condus la apariția de limbaje care să permită utilizarea ei în scrierea programelor. Limbajul C a fost și el dezvoltat în această direcție. Astfel, în anul 1980 s-a dat publicității *limbajul C++*.

Limbajul C++ a fost elaborat de Bjarne Stroustrup de la AT&T. El este un supersei al limbajului C și permite utilizarea principalelor concepte ale programării orientate spre obiecte.

Limbajul Pascal a fost și el dezvoltat în această direcție.

La ora actuală, majoritatea limbajelor de programare moderne au fost dezvoltate în direcția programării orientate spre obiecte.

În prezent, anii '80 sînt considerați ca fiind decada care a lansat programarea orientată spre obiecte.

Limbajul C++, ca și limbajul C, se bucură de o portabilitate mare și este implementat pe o gamă largă de calculatoare începînd cu microcalculatoare și pînă la cele mai mari supercalculatoare. Între diferitele implementări ale limbajului C++ există diferențe legate de specificul calculatoarelor, dar aceste diferențe nu sînt esențiale.

Limbajul C++ a fost implementat pe microcalculatoarele compatibile IBM PC în mai multe variante. Cele mai importante implementări ale limbajului C++ pe aceste calculatoare sînt cele realizate de firmele Microsoft și Borland. Aceste firme au implementat limbajul C++ pe microcalculatoarele respective în mai multe versiuni.

În cartea de față se prezintă facilitățile de bază ale limbajului C++ care sînt comune diferitelor implementări ale sale. Exercițiile prezentate în cartea de față au fost scrise folosind compilatorul TURBO C++ v2.0 (acest compilator este marcă înregistrată a firmei Borland).

Conceptele programării orientate spre obiecte au influențat în mare măsură dezvoltarea limbajelor de programare din ultimii ani. De obicei, multe limbaje au fost extinse așa încît ele să admită conceptele mai importante ale programării orientate spre obiecte. Uneori s-au făcut chiar mai multe extensii ale aceluiași limbaj.

De exemplu, în prezent asistăm la extensiile limbajului C++. O astfel de extensie o constituie *limbajul E* al cărui autori sînt Joel E. Richardson, Michael J. Carey și Daniel T. Schuh de la universitatea Wisconsin Madison.

Acest limbaj a fost proiectat pentru a permite exprimarea simplă a

tipurilor și operațiilor interne sistemelor de gestiune a bazelor de date.

Printre altele, limbajul E permite crearea și gestiunea *obiectelor persistente*, lucru deosebit de important pentru sistemele de gestiune a bazelor de date.

O altă extensie a limbajului C++ este *limbajul O* dezvoltat la Bell Laboratories. Cele două limbaje (E și O) sînt în esență echivalente.

Despre limbajul E se afirmă că este în principal orientat spre scrierea programelor sistem, în particular pentru implementarea sistemelor de gestiune a bazelor de date.

Despre limbajul O se spune că încearcă să îmbine facilitățile de nivel înalt cu cele ale programării sistem.

O altă extensie a lui C++ este limbajul Avalon/C++. Ea a fost realizată de Dettlefs D., Herlihy M. și Wing J. Acest limbaj este destinat pentru a suporta calcul distribuit.

Printre altele, aceste ultime două extensii admit și ele obiecte persistente.

În prezent, conceptele programării orientate spre obiecte au aplicații importante nu numai în ce privește dezvoltarea limbajelor de programare, ci și la implementarea altor sisteme complexe.

Cele mai mari realizări obținute pînă în momentul de față sînt legate de implementarea *sistemelor de gestiune a bazelor de date* și a *interfețelor utilizator*.

Limbajul E a fost utilizat, de către autorii lui, la implementarea proiectului EXODUS care este un sistem de gestiune a unei baze de date "extensibile".

În prezent există numeroase proiecte de acest fel implementate cu ajutorul conceptelor programării orientate spre obiecte. Bazele de date respective se numesc *baze de date orientate spre obiecte*.

Interfețele utilizator au atins o dezvoltare mare datorită facilităților oferite de componentele hardware ale diferitelor tipuri de calculatoare. În principiu ele simplifică interacțiunea dintre programe și utilizatorii acestora. Astfel, diferite comenzi, date de intrare sau rezultate pot fi exprimate simplu și natural utilizînd diferite standarde care conțin ferestre, bare de meniuri, cutii de dialoguri, butoane etc. Un dispozitiv primar de selectare utilizat foarte frecvent și cu mult succes este așa numitul "șoarece" (mouse). Toate acestea conduc la interfețe simple și vizuale, accesibile pentru utilizatorii finali.

Implementarea interfețelor este mult simplificată prin utilizarea limbajelor orientate spre obiecte. Aceasta mai ales datorită posibilităților de a utiliza componente standardizate aflate în biblioteci specifice.

Importanța aplicării conceptului de reutilizare prin intermediul limbajelor orientate spre obiecte rezulta din faptul că interfețele utilizator adesea ocupă 40% din codul total al aplicației.

Limbajele utilizate frecvent la implementarea interfețelor utilizator sînt limbajele C++ și Pascal.

Firma Borland comercializează o bibliotecă de componente standardizate care pot fi utilizate la implementarea interfețelor utilizator folosind unul din limbajele C++ și Pascal.

Produsul respectiv se numește *Turbo Vision*. Utilizarea lui pentru implementarea de interfețe utilizator presupune cunoașterea a cel puțin unuia din aceste limbaje de programare.

Interfețele utilizator implementate cu ajutorul limbajelor orientate spre obiecte le vom numi *interfețe utilizator orientate spre obiecte*.

De obicei, interfețele utilizator gestionează ecranul în mod grafic. O astfel de interfață utilizator se numește *interfață utilizator grafică*.

Una dintre cele mai populare interfețe utilizator grafice pentru calculatoarele compatibile IBM PC este produsul *Windows* oferit de firma Microsoft. Produsul a fost lansat în noiembrie 1985 și pînă acum a fost vîndut în cîteva milioane de exemplare. În prezent este lansată în exploatare versiunea v3.1.

Windows este un mediu de programare care amplifică facilitățile sistemului de operare MS-DOS. Aplicațiile Windows se pot dezvolta folosind diferite medii de dezvoltare ca: Turbo C++ pentru Windows, Pascal pentru Windows, Microsoft C++ 7.0, Microsoft Visual Basic, Visual C și Visual C++.

Componentele Visuale permit specificarea în mod grafic a interfeței utilizator, a unei aplicații, folosind șoarecele, iar aplicația propriu-zisă se programează într-un limbaj de tip Basic, C sau C++.

Din cele de mai sus, putem afirma că limbajul C++ se bucură de o popularitate crescută fiind utilizat în măsură mare la implementarea de aplicații complexe.

Dacă prin anii '70 se considera că o singură persoană este rezonabil să se ocupe cu programe de pînă la 4-5 mii de instrucțiuni, în prezent, în condițiile folosirii limbajelor de programare orientate spre obiecte, această medie este de 25 mii de instrucțiuni.



Cartea de față se adresează tuturor celor care doresc să se familiarizeze cu limbajul C++. Din motive de spațiu, în carte nu sînt date detalii cu privire la noțiunile de bază ale programării, arhitectura sau modul de funcționare al unui calculator, considerîndu-se că cititorul este familiarizat cu cel puțin un limbaj de programare. Tot din lipsă de spațiu nu este descris mediul de dezvoltare integrat TURBO C++ el fiind foarte asemănător cu mediile integrate de dezvoltare ale celorlalte compilatoare furnizate de firma Borland. Cititorul care nu este familiarizat cu astfel de medii poate consulta alte cărți care descriu astfel de medii (vezi de exemplu [17]).

În încheiere exprim mulțumiri D-nei Irina Mitrov pentru observațiile și sugeriile care mi le-a făcut în legătură cu conținutul cărții și al exercițiilor, precum și cu privire la îmbunătățirea clarității și exprimării diferitelor noțiuni. Menționez activitatea intensă depusă de D-na Irina Mitrov și fiica mea, Lavinia Negrescu, în legătură cu munca de introducere pe calculator a textului cărții. Tot pe această cale doresc să-i mulțumesc fiului meu, Dan Negrescu, pentru ajutorul pe care mi l-a dat în legătură cu formularea și rezolvarea unor exerciții care sînt legate de facilitățile sistemului de operare MS-DOS.

De asemenea, mulțumesc editurii Casa de Editură Albastră pentru efortul depus în vederea publicării îngrijite a cărții de față.

*Autorul*

# INTRODUCERE

Limbajul C++ este un superset al limbajului C avînd o utilizare universală și care permite scrierea de programe orientate spre obiecte.

Faptul că limbajul C++ este un superset al limbajului C înseamnă că orice program scris în limbajul C este în același timp și un program scris în limbajul C++. Această afirmație este adevărată cu mici excepții, de exemplu, compilatorul C++ face unele controale suplimentare față de compilatorul C. Aceste controale se referă în primul rînd la tipurile parametrilor funcțiilor, precum și la tipurile valorilor returnate de ele.

Faptul că limbajul C++ permite scrierea de programe orientate spre obiecte înseamnă că el permite utilizarea unor concepte de bază ale acestui stil de programare. Acest fapt reprezintă un salt calitativ față de limbajul C, ceea ce se exprimă sugestiv și prin numele limbajului (C++) El nu a fost numit limbajul D deoarece este o extensie a lui C.

Ținînd seama că operatorul "++", din limbajul C, reprezintă operația de incrementare, numele C++ se poate citi "C incrementat".

În principiu, putem afirma că limbajul C++ este un "C mai bun", care oferă atît facilitățile limbajului C cît și programarea orientată spre obiecte.

Facilitățile principale ale limbajului C sînt:

- portabilitatea mare a programelor;
- flexibilitatea în programare;
- programe compacte;
- lucrul pe biți;
- calcul de adrese.

Menționăm că limbajul C inițial a fost proiectat în ideea de a asigura o portabilitate bună a programelor.

Ca o consecință a acestui deziderat amintim lipsa, din limbajul C, a instrucțiunilor de intrare/ieșire. Aceste operații se realizează prin funcții de bibliotecă. Eliminarea din limbaj a instrucțiunilor de intrare/ieșire rezultă din faptul că operațiile de intrare/ieșire sînt dependente de particularitățile hardware ale calculatoarelor.

Flexibilitatea în programare rezultă, printre altele, din numărul mai redus de *controale* pe care îl face compilatorul C, față de alte compilatoare, de exemplu față de compilatorul limbajului Pascal. Acest fapt a provocat o serie de discuții, întrucît absența unor controale severe la compilare se

consideră că îngreunează depistarea unor erori în programare. Cu toate acestea, flexibilitatea în programare obținută în acest fel s-a dovedit a fi adesea utilă și trebuie considerată o facilitate în plus a limbajului decât o parte negativă a lui.

Alte facilități care măresc flexibilitatea în programare sunt legate de prezența în limbaj a unor instrucțiuni care asigură revenirea simplă dintr-o funcție, ieșirea din cicluri etc.

Compilatorul C++ realizează unele controale suplimentare față de compilatorul C. Cu toate acestea programarea în limbajul C++ se bucură de o flexibilitate mare.

O altă facilitate importantă a limbajului C este posibilitatea de a compacta programele sursă. Limbajul C permite o compactare bună a programelor sursă pe seama unor construcții specifice, cum sunt expresia de atribuire, expresia condițională, operatori de incrementare și decrementare etc.

Limbajul C oferă facilități specifice limbajelor de asamblare, cum sunt lucrul pe biți și calculul cu adrese. Acest fapt permite adesea scrierea de programe optime, atât în ceea ce privește memoria, cât și timpul de execuție.

Limbajul C se consideră că este un intermediar între limbajele de nivel înalt și cele de asamblare. El a câștigat o popularitate mare în decursul anilor datorită facilităților amintite mai sus.

La ora actuală există un standard ANSI al limbajului C, dar numeroasele sale implementări se abat de la acest standard.

În cartea de față ne vom referi la versiunea TURBO C v2.0 care este marcă înregistrată a firmei BORLAND și este implementată pe calculatoarele compatibile IBM PC.

Compilatorul TURBO C poate fi apelat prin intermediul *mediului integrat de dezvoltare* (Integrated Development Environment) sau printr-o linie de comandă.

Iară a intra în detaliu cu privire la mediul integrat de dezvoltare TURBO C, indicăm mai jos o secvență simplă de comenzi pentru a lansa în execuție un program C nou:

- după setarea pe directorul corespunzător se tastează
  - tc;
  - <Alt>-F;
  - N.
- se editează programul sursă folosind editorul mediului TURBO C;

- se acționează tasta F2;
- se indică numele fișierului pentru salvarea programului sursă pe disc;
- se realizează compilarea, link-editarea și lansarea în execuție a programului tastând:

<Ctrl>-F9

- pentru a vizualiza rezultatele execuției programului se tastează:

<Alt>-F5

- se revine în fereastra de editare a mediului acționând o tastă oarecare.

Tasta F1 poate fi acționată pentru a obține informații suplimentare. Se revine în tasta precedentă acționând tasta ESC.

Pentru a ieși din mediul integrat de dezvoltare TURBO C se poate tasta.

<Alt>-X

Compilatorul TURBO C++ poate fi și el apelat prin intermediul mediului integrat de dezvoltare TURBO C++ sau printr-o linie de comandă.

Mediul integrat de dezvoltare TURBO C++ este asemănător cu cel al compilatorului TURBO C. De exemplu, secvența de mai sus poate fi utilizată și în cazul compilatorului TURBO C++ v2.0 cu singura diferență că în loc de tasta F2 se tastează F9. Menționăm că fișierele cu programe sursă C au extensia .C iar cele cu programele sursă C++ au extensia .CPP. Programele sursă C din această carte au fost compilate cu compilatorul C++ și ele au extensia .CPP.

Limbaajul C++ este de limbaajul Pascal și celelalte limbaaje precedente lor sînt adecvate *programării procedurale*.

Programarea procedurală este un model de programare utilizat frecvent încă din ziua înălță de existență a limbaajelor de programare.

La baza acestei programări se află *procedura*. Aceasta poate fi considerată ca un proces de *abstracție*. Acest proces se realizează pe baza *parametrilor* care permit realizarea procedurii făcînd *abstracție de valorile* care sînt necesare pentru care să se execute procedura.

De exemplu, la realizarea unei proceduri pentru calculul funcției *sin* se face abstracție de valoarea concretă a unghiului. Procedura are un parametru care ne permite să neglijăm valoarea concretă a unghiului. Procedura se definește în așa fel încît ea să calculeze valoarea funcției *sin* în

funcție de valoarea parametrului ei, valoare care se concretizează la execuție în momentul apelului procedurii respective.

Acest proces de abstractizare prin intermediul parametrilor procedurilor se numește *abstractizare procedurală*.

Un alt aspect al acestui proces de abstractizare este acela că procedura poate fi privită ca o "cutie neagră". La apelul ei nu se au în vedere detaliile de realizare ale procedurii. De exemplu, la apelul procedurii de calcul a valorii funcției *sin* sînt neglijate detaliile cu privire la algoritmul de calcul al funcției respective. La aceasta fază interesează "tipul" parametrului și eventual precizia de calcul a funcției respective.

Abstractizarea procedurală a fost folosită pe scară largă la rezolvarea multor probleme aplicative sau de sistem. Importanța ei crește odată cu creșterea complexității problemei de rezolvat. De obicei, problemele complexe se descompun în subprobleme mai simple. Acestea, la rîndul lor, pot fi și ele descompuse în continuare și așa mai departe, pînă cînd se ajunge la componente suficient de simple. Acest proces de descompunere se realizează printr-un proces de abstractizare care permite neglijarea diferitelor detalii. În felul acesta pot fi puse în evidență procese de calcul care prin utilizări de parametri pot fi exprimate printr-o procedură.

Pasul următor programării procedurale a fost *programarea modulară*. Deși nu există o definiție riguroasă pentru noțiunea de modul, citîndu-l pe Bjarne Stroustrup, se poate afirma că *modulul* este un set de proceduri înrudite împreună cu datele pe care le manevrează (vezi [14]).

Ceea ce trebuie să mai precizăm este faptul că datele manevrate de aceste proceduri, care compun un modul, sînt de obicei "ascunse" în modulul respectiv. Aceasta înseamnă că în afara modulului nu se are acces direct la datele ascunse de modul. Se are acces la ele numai indirect, prin intermediul procedurilor modulului respectiv. Limitarea accesului la date prin ascunderea lor într-un modul conduce la o protejare a lor, ceea ce prezintă importanță mai ales în cazul programelor complexe.

Un exemplu simplu de modul este modulul pentru implementarea unei *stive*. În forma cea mai simplă, stiva poate fi considerată ca fiind o zonă de memorie în care pot fi pastrate date, de obicei de un același tip. Datele pot fi scoase din stivă numai în ordinea inversă păstrării lor. Ultima dată pusă în stivă se spune că este în vîrfurile ei. Data scoasă din stivă este totdeauna aceea aflată în vîrfurile stivei. Totodată, după scoaterea unei date din stivă, în vîrfurile stivei se va afla data care a fost pusă pe stivă imediat înaintea celei scoase din stivă. În felul acesta, totdeauna se scoate din stivă ultima dată pastrată (pusă) în stivă. Se obișnuiește să se spună că stiva este o zonă de memorie gestionată după principiul: ultimul pus pe stivă este primul scos din stivă.

(last in-first out sau pe scurt LIFO). Este important ca acest principiu să fie respectat. De exemplu, nu este admis să se scoată din stivă sau să se aibă acces la un element care nu este în vârful stivei. De asemenea, când se pune un element pe stivă, acesta poate fi pus numai după cel aflat în vârful stivei și prin aceasta el devine elementul *din* vârful stivei. Din această cauză, zona de memorie prin care se implementează stiva trebuie "ascunsă", accesul la ea realizându-se numai prin intermediul a două proceduri:

- una care să pună o dată pe stivă (procedură care de obicei se numește *push*);

și

- una care să scoată o dată din stivă (procedură care de obicei se numește *pop*).

De obicei, la aceste proceduri se mai adaugă o procedură pentru inițializarea stivei. Această procedură are drept scop vidarea stivei. În urma apelului ei, stiva devine vidă (această procedură de obicei se numește *clear*).

Cele trei proceduri amintite mai sus se consideră "inrudite" și împreună cu zona de memorie utilizată pentru implementarea stivei se grupează într-un modul. Utilizatorul nu poate gestiona direct această zonă de memorie decât numai prin intermediul celor 3 proceduri amintite mai sus.

Un modul similar se poate defini pentru a gestiona o zonă de memorie conform principiului: primul pus în zona respectivă este și primul scos (first in-first out sau pe scurt FIFO). O astfel de zonă de memorie se spune că formează o *coadă*.

Adesea, ascunderea datelor în modul se extinde și asupra procedurilor, limitându-se accesul din afara modului la unele proceduri din compunerea lui.

Programarea modulară corespunde mai bine ideii amintite mai sus cu privire la descompunerea problemelor complexe în subprobleme mai simple. Ea permite un grad mai înalt de abstractizare pe seama ascunderii datelor și a procedurilor.

Limbajul C a fost proiectat în așa fel încât să permită realizarea de module cu ascunderea datelor și a procedurilor. Astfel, datele și procedurile declarate cu *static* au o valabilitate limitată în cadrul fișierului sursă în care sint declarate.

Programarea modulară are anumite limite. Așa de exemplu, dacă este nevoie de mai multe stive, atunci programarea modulară este o soluție destul de complexă pentru a gestiona stivele respective.



O rezolvare mai simplă și naturală s-ar obține dacă am putea defini date de tip stivă, așa cum putem defini, de exemplu, date de tip întreg, flotant sau caracter.

Toate limbajele de nivel înalt pun la dispoziția utilizatorului un anumit număr de tipuri predefinite. De exemplu, limbajul FORTRAN are tipurile predefinite *integer* și *real*. Limbajul C are și el aceste tipuri predefinite (întreg și flotant) și în plus mai pune la dispoziția utilizatorului tipul caracter.

Un *tip de date* descrie un set de date pe care le mai numim și obiecte, care au aceeași reprezentare. De asemenea, există un număr de operații asociat cu un tip de date. De exemplu, cu tipul întreg se asociază cele patru operații aritmetice și eventual și altele.

Ulterior s-a constatat că este util ca limbajul să permită utilizatorului să definească el însuși tipuri diferite de cele predefinite în limbaj. În acest scop, în limbajul Pascal s-a introdus noțiunea de *înregistrare* (record) care permite definirea de grupe de date care nu neapărat au un același tip. O astfel de grupă de date reprezintă un tip nou de date. O facilitate analogă există și în limbajul C. În acest caz se utilizează noțiunea de *structură* (struct). Altfel construcția record din Pascal și struct din limbajul C definesc *reprezentarea tipului nou de dată*.

Utilizatorul definește, de asemenea, operații cu date de acest tip prin intermediul unor proceduri sau funcții.

Tipurile definite în acest fel se numesc *tipuri definite de utilizator* sau mai scurt *tipuri utilizator*. Un exemplu de tip utilizator folosit frecvent poate fi tipul de număr complex. Acesta are o reprezentare care se definește simplu în limbajul C folosind construcția struct. Numărul complex este o grupă formată din două numere flotante, primul reprezintă partea reală a numărului complex, iar celalalt partea imaginară a lui.

În continuare, se pot defini variabile care să aibă tipul complex, exact la fel cum se definesc variabile de tipul întreg, flotant sau caracter.

Pentru a realiza operații cu numere complexe, utilizatorul definește funcții corespunzătoare. Operațiile care se execută asupra datelor care au tipuri predefinite au diferite facilități care lipsesc în cazul tipurilor utilizator. Așa de exemplu, pentru datele de tip întreg sau flotant se utilizează operatorii obișnuiți pentru operațiile aritmetice (+, -, \* și /). Acest lucru nu este posibil și pentru numerele de tip complex. De asemenea, o serie de controale și conversii se realizează automat în cazul utilizării datelor de tipuri predefinite.

Un alt neajuns al tipurilor utilizator, definite ca mai sus, este faptul că nu

se asigură nici o protecție asupra componentelor unei date

Aceste neajunsuri rezultă mai ales din faptul că la definirea tipurilor utilizator de fapt se definesc numai reprezentările datelor respective. Nu se precizează încălări operațiile posibile asupra datelor respective. Chiar dacă astfel de operații sînt definite prin proceduri sau funcții, compilatoarele nu au informații în acest sens. De aceea, pasul următor constă în posibilitatea de a preciza în limbaj procedurile sau funcțiile care au acces la componentele unei date de tip utilizator. O astfel de extensie s-a realizat foarte simplu, de exemplu în cazul construcției struct din limbajul C, alături de definirea reprezentării componentelor se indică și lista funcțiilor care definesc operațiile asupra lor. Aceasta însă, nu înseamnă că numai funcțiile respective au acces la datele componente ale structurii. Cu alte cuvinte, prin enumerarea funcțiilor în cadrul construcției struct nu se oferă nici o protecție a componentelor structurii respective, ci numai se indică faptul că datele componente ale structurii sînt prelucrate prin funcțiile enumerate, dar la ele putem avea acces și prin alte funcții.

Mai tîrziu s-a constatat că adesea anumite componente date și eventual și componente funcții atașate unui tip definit de utilizator este bine să fie protejate pentru a înlătura accesul neautorizat. În felul acesta s-a ajuns la noțiunea de *tip abstract de dată*. Aceasta noțiune depășește posibilitățile limbajului C. Tipurile abstracte de date pot fi definite în limbajul C++ folosind noțiunea de *clasă*. O clasă, deci, definește atât reprezentarea datelor tipului respectiv cît și funcțiile care au acces și pot prelucra datele respective.

La definirea unei clase se indică componentele dată și funcție la care au acces utilizatorii clasei (componente *publice*). Componentele care nu sînt publice permit un acces limitat și în felul acesta ele sînt protejate față de accesurile neautorizate. Se obișnuiește să se spună că, clasele "încapsulează" datele. Acest proces de încapsulare este deosebit de important permițînd definirea de componente *protejate și reutilizabile*. Deosebirea dintre clasă și construcția struct constă chiar în ideea de protecție. Un tip utilizator definit printr-o construcție struct poate fi definit printr-o clasă care are toate componentele dată și funcție, publice.

O dată de un tip abstract (adică tip definit printr-o clasă) se spune că este un *obiect*. De asemenea, se obișnuiește să se spună că obiectul este o *instanțare* a clasei care deținește tipul său.

Reluînd exemplul cu privire la definirea și gestiunea stivelor, observăm că de data aceasta putem introduce tipul abstract de dată *stivă* prin intermediul unei clase. Zona de memorie care se organizează ca stivă este o componentă dată protejată a clasei respective. O altă componentă dată protejată este cea care definește virful stivei. Componentele funcție,

publice, ale clasei sînt funcțiile *push*, *pop* și *clear*. Ele pot fi apelate din diferite funcții ale programului. Deoarece componentele date sînt protejate, la ele nu se are acces decît prin intermediul celor trei funcții publice indicate mai sus. Prin instanțierea acestei clase se obține un obiect de tip stivă. În felul acesta se pot obține atîtea obiecte de tip stivă, cîte sînt necesare, exact la fel cum se pot defini atîtea date de un tip predefinit, cîte sînt necesare.

În mod analog, se poate defini, printr-o clasă, tipul abstract de dată pentru numere complexe. În acest caz sînt necesare două componente dată, una pentru partea reală și una pentru partea imaginară. Ambele componente sînt protejate. Cele patru operații aritmetice asupra numerelor complexe se definesc prin patru componente funcție care sînt publice.

Un număr complex este o instanțiere a acestei clase. Se pot instanția atîtea numere complexe cîte sînt necesare.

Mai mult decît atît, componentele funcție care definesc cele patru operații aritmetice cu numere de tip complex pot fi definite în așa fel încît operatorii acestor operații să poată fi folosiți nu numai pentru date de tipuri predefinite ale limbajului, ci și pentru datele de tip complex. Așa de exemplu, funcția pentru adunarea a două numere complexe poate fi definită în așa fel ca expresia  $x+y$  să fie valabilă nu numai pentru cazurile în care  $x$  și  $y$  sînt de tip întreg sau flotant, ci și cînd ei sînt de tip complex.

Cu alte cuvinte, este posibil ca valabilitatea de aplicare a unui operator al limbajului să fie extinsă și pentru tipuri abstracte de date. Această posibilitate trebuie privită ca un mijloc de extindere a valabilității operatorilor existenți. Acest mijloc îl vom numi în continuare *supraîncărcarea operatorilor*.

Menționăm că, clasele au de obicei și alte componente funcție care permit realizarea unor operații specifice tipurilor abstracte de date. De exemplu, obiectele pot fi inițializate la instanțierea lor printr-o componentă funcție specială, care se numește *constructor*.

De asemenea, o altă componentă funcție specială realizează "distrugearea" obiectului. Ea se numește *destructor*.

Alte componente funcție se definesc pentru a copia obiecte, a face diferite conversii etc.

Toate aceste facilități apropie mult comportamentul obiectelor instanțiate prin clase de cele care au tipuri predefinite.

Programarea care utilizează tipuri abstracte de date este un stil de programare superior programării modulare. Un astfel de stil de programare se numește *programare prin abstracțizarea datelor*. În esență, ea constă în definirea de tipuri abstracte de date pentru fiecare concept necesar la

rezolvarea unei probleme concrete, concept care nu este predefinit în limbajul de programare utilizat (mai sus s-au indicat conceptele de stivă și număr complex). În felul acesta programarea devine mai simplă și mai naturală.

De aici și pînă la stilul programării orientate spre obiecte mai este un singur pas.

De obicei, diferite tipuri abstracte de date au elemente comune. Precizarea lor conduce la o "ierarhizare" a tipurilor abstracte de date. În virful ierarhiei se află clasa ce conține elementele comune celorlalte clase. Ea este clasa cea mai generală. Pe nivelul următor al ierarhiei se află alte clase care conțin elementele clasei din virful ierarhiei, precum și alte componente specifice tipurilor abstracte pe care le definesc. În general, dacă o clasă conține și alte componente decît cele care sînt comune pentru una sau mai multe clase, atunci aceasta se află la un nivel mai inferior al ierarhiei decît clasa care definește elementele comune lor.

Ierarhizarea claselor are o importanță mare, deoarece atributele unei clase rămîn valabile pentru clasele de pe nivelul următor ei din ierarhie. Această proprietate se numește *moștenire*.

Pe scurt, moștenirea permite definirea de tipuri abstracte de date noi prin adăugarea de componente dată și/sau funcție la un tip abstract de date deja existent. Prin aceasta se pot elabora simplu componente extensibile.

O clasă care se definește adăugînd componente noi unei clase date se numește *clasă derivată*, iar clasa compusă din elementele comune (la care s-au adăugat componentele noi) se numește *clasă de bază*. Această terminologie este utilizată în C++.

Uneori clasa derivată se mai numește și subclasă a clasei de bază, iar clasa de bază se numește superclasă a unei clase derivate a ei.

Noțiunea de moștenire se aplică cu succes la prelucrarea conceptelor ierarhice din lumea reală.

Modelarea ierarhiilor din lumea reală conduce la o ierarhie de tipuri abstracte de date care în limbajul C++ se definește printr-o ierarhie de clase bazată pe proprietatea de moștenire

Un exemplu simplu de ierarhie din lumea reală este o comunitate de persoane împreună cu conducătorii acestei comunități. În acest exemplu simplu, există două concepte și anume conceptul *persoană* și conceptul de *conducător*.

Conceptul de persoană se modelează printr-o clasă care conține datele unei persoane:

- nume;
- prenume;
- data naşterii;
- localitate;
- domiciliu;
- profesiune;
- studii;
- stare civilă;
- număr copii;
- încadrare;
- vechime în muncă;
- salarizare;
- cod.

Conceptul de conducător derivă din cel de persoană deoarece orice conducător are toate atribuţiile specifice unei persoane. De aceea, conceptul de conducător se modelează printr-o clasă care derivă din clasa persoană.

Conceptul de conducător presupune elemente noi, cum ar fi de exemplu cunoştinţe despre conducere, specializare, funcţie etc.

Un alt exemplu simplu este ales din geometrie şi se referă la patrulater.

Dacă se consideră clasa care defineşte conceptul de paralelogram (patrulater cu laturile opuse paralele), atunci conceptul de dreptunghi poate fi modelat printr-o clasă derivată din cea a paralelogramului. Într-adevăr, dreptunghiul este un paralelogram la care mai adăugăm condiţia ca să aibă un unghi drept. Deci clasa dreptunghi este o clasă derivată a clasei paralelogram iar aceasta din urmă este o clasă de bază a clasei dreptunghi. Ierarhia poate fi continuată cu pătratul. Într-adevăr, pătratul este un dreptunghi cu toate laturile egale, deci clasa corespunzătoare pătratului este derivată din cea a dreptunghiului. De asemenea, clasa pentru dreptunghi este clasa de bază pentru clasa corespunzătoare pătratului.

Deoarece rombul este un paralelogram cu toate laturile egale, clasa corespunzătoare acestuia este şi ea o clasă derivată din clasa corespunzătoare paralelogramului. Deci, clasa pentru paralelogram este clasa de bază atât pentru clasa corespunzătoare dreptunghiului cât şi pentru cea corespunzătoare rombului.

La rândul ei, clasa corespunzătoare paralelogramului poate fi şi ea derivată dintr-o clasă mai generală corespunzătoare patrulaterelor.

Numărul exemplurilor poate fi mărit ușor, deoarece lumea reală este plină de ierarhii. De aici decurge importanța mare a conceptului de moștenire care distanțează substanțial stilul programării orientate spre obiecte de cel al programării prin abstractizarea datelor.

În concluzie programarea procedurală este cea mai veche și ea se află la nivelul cel mai de jos din punctul de vedere al stilului de programare. La nivelul următor se află programarea modulară care permite ascunderea datelor și a procedurilor în module. Ea este suficientă pentru programarea problemelor relativ simple care nu implică utilizări multiple ale conceptelor modelate prin module. De exemplu, dacă conceptul de stivă este realizat printr-un modul, atunci programarea modulară este suficientă dacă în program nu se utilizează în același timp mai multe stive de felul celei definite prin modulul respectiv.

Ambele stiluri (programarea procedurală și cea modulară) sunt suportate de limbajul C.

Nivelul următor, programarea prin abstractizarea datelor, înlătură inconvenientele programării modulare pe baza modelării conceptelor ce intervin în rezolvarea problemelor prin tipuri abstracte de date. De exemplu, introducând tipul abstract de date "stivă", se pot defini și utiliza simplu mai multe stive simultan în același program.

Programarea prin abstractizarea datelor nu este suportată de limbajul C.

Nivelul următor îl constituie programarea orientată spre obiecte. Acest stil de programare, pe lângă faptul că permite modelarea conceptelor prin tipuri abstracte de date, ea permite în plus și alte facilități dintre care mai sus s-a amintit exprimarea ierarhiilor din cadrul conceptelor prin facilitatea de moștenire.

Programarea prin abstractizarea datelor este suficientă pentru probleme în care intervin concepte individuale care nu au elemente comune și care deci nu conduc la existența unei ierarhii.

Limbajul C++ a fost proiectat pentru a permite definirea și utilizarea tipurilor abstracte de date, precum și ierarhizarea lor prin folosirea conceptului de moștenire.

Programarea orientată spre obiecte implică și alte facilități care însă nu sunt suportate direct de limbajul C++. De altfel, la ora actuală nu există un limbaj care să suporte toate conceptele programării orientate spre obiecte.

În cartea de față se descriu elementele de bază ale limbajului C++ și totodată se dau exemple de utilizare a lor în programare.

Cartea apare în trei volume intitulate:



Limbajul C - volumul 1;

Limbajul C++ - volumul 2;

Limbajele C și C++ în aplicații - volumul 3.

În primul volum se face o descriere elementară a limbajului C. Această descriere este însoțită de o serie de exemple și programe simple care permit o mai bună înțelegere și fixare a elementelor limbajului C.

Majoritatea exemplelor au un caracter didactic. Cu toate acestea, cititorul va întâlni unele exemple și programe sau funcții care pot interveni în practica de zi cu zi a programării în limbajul C.

Autorul insistă asupra importanței rulării programelor prezentate în carte și propune ca ele să fie rulate și înțelese pe măsura ce se avansează cu citirea cărții.

Programele din primul volum pot fi compilate și executate folosind medile integrate de dezvoltare Turbo C sau Turbo C++ implementate de firma BORLAND.

În cazul în care se folosește mediul Turbo C se va folosi extensia .C la fișierele sursă, iar în cazul lui Turbo C++ extensia .CPP.

Programele din volumul 2 se pot compila și executa numai sub mediul Turbo C++ și fișierele sursă vor avea extensia .CPP.

În volumul 3 se întâlnesc atât programe care nu utilizează elemente specifice limbajului C++ cât și programe care utilizează astfel de elemente. Programele din prima categorie pot fi compilate și executate folosind ambele medii amintite mai sus. Evident cele din categoria a doua implică utilizarea mediului Turbo C++.

Volumul 2, conține o descriere elementară a facilităților oferite de limbajul C++, care este însoțită de numeroase exemple și programe menite să faciliteze înțelegerea și fixarea cunoștințelor respective.

Abordarea programelor din volumul 3 este posibilă numai după însușirea temeinică a cunoștințelor și facilităților limbajelor C și C++ prezentate în primele două volume ale cărții.

Autorul recomandă programatorilor începători să parcurgă volumele 1 - 2 de mai multe ori înainte de a trece la studierea programelor din volumul 3.

Limbajele C și C++ având un caracter universal, volumul 3 conține programe pentru rezolvări de probleme de sistem, probleme orientate spre calcule științifice sau prelucrări de date.

Limbajul C se utilizează în aplicații care nu implică elemente specifice programării orientate spre obiecte:

- tipuri abstracte de date;
- moștenire;
- supraîncărcarea operatorilor etc.

Majoritatea problemelor de acest fel sînt orientate spre calcule științifice, ca de exemplu:

- rezolvări de ecuații;
- sisteme de ecuații liniare;
- probleme de optimizări etc.

În schimb, aplicațiile de sistem și de grafică se abordează utilizînd facilitățile programării orientate spre obiecte

# 1. NOȚIUNI DE BAZĂ

Un program conține una sau mai multe *funcții*. Dintre acestea, una este funcția *principală*.

Fiecare funcție are un *nume*. Numele funcției principale este *main*. Celelalte funcții au nume definite de utilizator.

Programul se păstrează într-un *fișier* sau mai multe. Fișierele au extensia *.c* pentru limbajul C și *.cpp* pentru limbajul C++.

Un fișier care conține un program scris în C sau C++ sau care conține numai o parte a acestuia se va numi *fișier sursă*. Prin compilarea unui fișier sursă rezultă un *fișier obiect*. Acesta are extensia *.obj*.

Fișierele sursă care intră în compunerea unui program pot fi compilate împreună sau separat. În urma unei compilări rezultă un fișier obiect. Fișierele obiect, corespunzătoare unui program, pot fi reunite într-un program executabil prin *edita de legături* (*link-editare*). În urma link-editării rezultă un *fișier executabil*. Acesta are extensia *.exe*.

## 1.1. Nume

Un *nume* este o succesiune de litere și eventual și cifre, primul caracter fiind literă. În calitate de *litere* se pot utiliza *literele mici* și *mari* ale alfabetului englez, precum și caracterul *subliniere* (*\_*).

Numărul de caractere care intră în compunerea unui nume nu este limitat.

În mod implicit, numai primele 32 de caractere dintr-un nume sînt luate în seamă. Aceasta înseamnă că două nume diferă între ele numai dacă ele diferă în primele 32 de caractere ale lor.

Menționăm că sistemele integrate de dezvoltare Turbo C și Turbo C++ permit utilizatorului să modifice această limită de 32.

**Exemple de nume:**

```
x  
i  
a1  
a_1  
a1b2c3  
AxY  
_ (caracterul subliniere)
```

```
_1  
a_  
acesta_este_un_nume  
Acesta_Este_Tot_un_NUME
```

Se recomandă ca numele să fie sugestiv, adică el să sugereze pe cât posibil scopul alegerii lui sau a datei pe care o reprezintă.

La scrierea numelor se folosesc frecvent literele mici. Adesea, cînd un nume se formează din concatenarea mai multor prescurtări de cuvinte, fiecare cuvînt începe în numele respectiv cu o literă mare.

**Exemple:**

```
prodScal sau ProdScal - pentru produs scalar;  
prodMat sau ProdMat - pentru produsul a două matrice.
```

## 1.2. Cuvinte cheie

Există un număr de cuvinte împrumutate din limba engleză care au o utilizare predefinită. Utilizatorul nu poate să dea o altă utilizare acestor cuvinte. Ele se numesc *cuvinte cheie*.

Cuvintele cheie se scriu cu litere mici. Deci cuvintele cheie sînt nume cu destinații speciale.

**Exemple:**

```
if  
while  
for  
break  
class etc.
```

Sensul cuvintelor cheie va fi explicat pe măsură ce se vor descrie construcțiile în care ele apar.

## 1.3. Tipuri de date de bază

Orice limbaj de programare oferă programatorului un număr de tipuri de date de bază.

Tipurile de bază se specifică prin cuvinte cheie. Acestea, în ordine alfabetică sînt:

```
char  
double
```

float  
int  
long  
short  
signed  
unsigned

În tabela de mai jos se indică reprezentarea tipurilor de bază.

**Reprezentarea tipurilor de bază - Tabela 1**

Specificarea tipului	Dimensiune în biți	Modul de reprezentare
int	16	întreg reprezentat prin complement față de doi
short	16	idem
long	32	idem
unsigned	16	întreg fără semn
unsigned long	32	idem
char	8	codul ASCII al caracterului
float	32	reprezentare flotantă în simplă precizie
double	64	reprezentare flotantă în dublă precizie
long double	80	reprezentare flotantă în dublă precizie

Datele de tip caracter pot fi specificate prin:

unsigned char

sau

signed char

În primul caz, data se presupune că este un întreg în intervalul [0,255] (întreg fără semn).

În cel de al doilea caz, data se presupune că aparține intervalului [-128,127] (întreg cu semn).

Datele specificate numai prin *char* au o interpretare implicită. Această interpretare poate fi definită prin intermediul mediului integrat de dezvoltare Turbo C sau Turbo C++. În mod normal se consideră ca datele de tip *char* sînt implicit numere fără semn.

Pentru a preîntîmpina apariția unor erori prin interpretări implicite nedorite ale datelor de tip caracter, se recomandă a se utiliza specificările explicite *unsigned char* și *signed char*.

În tabela 2 se indică intervalele de valori ale tipurilor de bază.

**Intervalele de valori pentru tipurile de bază - Tabela 2**

Specificarea tipului	Intervalul de valori
int	[-32768,32767]
short	[-32768,32767]
long	[-2147483648,2147483647]
unsigned	[0,65535]
unsigned long	[0,4294967295]
unsigned char	[0,255]
signed char	[-128,127]
float	valoarea absolută a unei date diferite de zero aparține intervalului $[3,4 \cdot 10^{**}(-38); 3,4 \cdot 10^{**}(38)]$
double	valoarea absolută a unei date diferite de zero aparține intervalului $[1,7 \cdot 10^{**}(-308); 1,7 \cdot 10^{**}(308)]$
long double	valoarea absolută a unei date diferite de zero

Specificarea tipului	Intervalul de valori
	aparține intervalului [3,4*10**(-4932);1,1*10**(4932)]

**Observație:**

Tipul predefinit *short* este identic cu tipul predefinit *int* la calculatoarele compatibile IBM PC. La alte calculatoare ele pot fi diferite, de exemplu tipul *short* rămâne reprezentat pe 16 biți, iar tipul *int* poate să fie la fel ca și tipul *long*.

Tipul *unsigned long* poate fi scris inversând cuvintele cheie:

*long unsigned*.

## 1.4. Structura unei funcții

O funcție are următoarea structură:

```
tip_nume (lista_declarărilor_parametrilor_formali)
{
    declarații
    instrucțiuni
}
```

Primul rînd din formatul de mai sus reprezintă *antetul* funcției. Partea inclusă între acolade, împreună cu acoladele, formează *corpul* funcției.

În cazul tipurilor predefinite, *tip* din antetul funcției este un cuvînt cheie. El definește tipul valorii returnate de funcție.

În limbajul C există două categorii de funcții. O primă categorie conține funcțiile care la revenirea din ele returnează o valoare în punctul de apel. Tipul acestei valori se definește prin tip din antetul funcției. Cealaltă categorie de funcții conține funcțiile care nu returnează nici o valoare la revenirea din ele.

Pentru aceste funcții se va folosi cuvîntul cheie *void* în calitate de *tip*. El semnifică lipsa unei valori returnate la revenirea din funcție.

O funcție poate avea zero sau mai mulți parametri. Lista declarațiilor parametrilor formali este vidă în cazul în care funcția nu are parametri formali. În acest caz antetul funcției se reduce la

*tip nume()*

Menționăm că absența parametrilor poate fi indicată explicit folosind cuvântul cheie *void*. Astfel, antetul de mai sus poate fi scris și sub forma:

*tip nume(void)*

**Exemple:**

1. **void f(void)**

```
{  
  ...  
}
```

Funcția *f* nu are parametri. Ea nu returnează nici o valoare la revenirea din ea.

2. **void f()**

```
{  
  ...  
}
```

Format identic cu cel din exemplul precedent.

3. **int g()**

```
{  
  ...  
}
```

Funcția *g* nu are parametri. La revenirea din ea se returnează o valoare întreagă de tip *int*.

4. **double h(void)**

```
{  
  ...  
}
```

Funcția *h* nu are parametri. La revenirea din ea se returnează o valoare flotantă în dublă precizie.

În cazul în care funcția are parametri, declarațiile parametrilor formali se includ între parantezele rotunde prezente după numele funcției și se separă prin virgulă dacă sînt mai multe.

Parametri se utilizează pentru a permite transferuri de date la o funcție în momentul apelului ei. Acest mecanism de transfer al datelor prin



intermediul parametrilor ne permite construirea de funcții făcând abstracție de valorile concrete care vor fi prezente abia la execuția programului. În momentul compilării este necesară numai cunoașterea *tipurilor* valorilor pe care le vor primi parametri la execuție. Aceste tipuri sînt definite prin declarațiile parametrilor respectivi, declarații care, așa cum am văzut mai sus, se indică în antetul funcției.

Parametri declarați în antetul unei funcții și care apoi se utilizează în corpul funcției, se numesc *formali* pentru a sublinia faptul că ei nu reprezintă valori concrete, ci numai în locul acestora pentru a putea exprima procesul de calcul realizat prin funcție. Ei se concretizează la execuție prin apelurile funcției. Valorile parametrilor formali se definesc la fiecare apel al unei funcții prin așa numiți parametri *reali, efectivi* sau *concreți*.

Utilizarea parametrilor formali la implementarea funcțiilor și atribuirea de valori concrete pentru ei la execuție, reprezintă un prim nivel de abstractizare în programare. Acest mod de programare este cel mai vechi. El se numește *programare procedurală* și realizează un proces de *abstractizare prin parametri*.

Declarațiile parametrilor formali sînt asemănătoare cu cele ale variabilelor și vor fi definite într-un paragraf ulterior.

#### Observații:

1. Inițial antetul unei funcții a avut următorul format:

*tip* nume(*lista parametrilor formali*)

*declarațiile parametrilor formali*

unde *tip* este prezent numai pentru funcții care returnează o valoare la revenirea din ele. Cuvîntul cheie *void* a fost introdus ulterior. De asemenea, cuvîntul cheie *int* nu era nevoie să fie prezent, considerîndu-se că orice funcție care returnează o valoare a cărui tip nu este specificat, returnează o valoare de tip *int*.

Ulterior s-a constatat că această libertate în omiterea tipului este o sursă de erori și de aceea se recomandă ca tipul să fie totdeauna prezent în antetul unei funcții. În cazul limbajului C++ contrarele cu privire la tipul valorii au fost întărite chiar pentru a elimina posibilitățile de apariție a erorilor la revenirea din funcții.

Lista parametrilor formali este fie vidă, cînd funcția nu are parametri, fie se compune dintr-un nume sau mai multe separate prin virgulă. În acest caz, declarațiile parametrilor formali, dacă ei există, se dau imediat după

paranteza închisă. Acest format poate fi folosit, atât în limbajul C, ca și în limbajul C++, dar compilatorul C++ remarcă printr-un avertisment că acest format este învechit.

De asemenea, amintim că cele două formate pot fi folosite împreună, adică putem indica în parantezele rotunde o parte din parametri prin numele lor, iar restul prin declarațiile lor.

În continuare se dau declarațiile parametrilor care în parantezele rotunde au fost prezente numai prin numele lor.

Cu toate acestea, se recomandă utilizarea formatului în care toate declarațiile parametrilor sînt incluse în parantezele rotunde (formatul de la începutul paragrafului).

2. Pentru funcția principală se pot utiliza antetele

```
int main()
int main(void)
void main()
void main(void)
    main()
    main(void)
```

Primele două antete presupun că funcția *main* returnează o valoare întreagă la revenirea din ea în sistemul de operare.

Adeesea se obișnuiește utilizarea formatului fără tip

```
main()
```

Amintim că funcția *main* poate avea și parametri. Aceștia, cînd sînt prezenți, permit utilizarea de către program a unor valori definite la lansarea programului. Ulterior vom preciza modul de utilizare al acestor parametri.

## 1.5. Comentarii

În limbajul C, ca și în alte limbafe de programare se pot folosi comentarii.

Un comentariu începe cu succesiunea de caractere

```
/*
```

și se termină cu

\*/

El se compune din orice caracter admis în setul de caractere al limbajului. Evident, în interiorul unui comentariu nu se va folosi succesiunea

\*/

care termină un comentariu.

În limbajul C++ s-a mai introdus o convenție pentru a insera comentarii. Astfel, în C++ un comentariu poate începe prin succesiunea

//

Un astfel de comentariu se termină pe același rând (la sfârșitul lui) pe care se află și începutul lui.

Comentariile sînt explicații pentru programatori. Ele nu au nici un efect asupra compilatorului și sînt omise la compilare.

Un comentariu se poate insera oriunde în program unde este legal să apară un spațiu, un tabulator sau caracterul de rînd nou.

În general, se recomandă introducerea de comentarii după antetul funcției, care să precizeze:

- acțiunea sau acțiunile realizate de funcție;
- formatele datelor de intrare și ieșire;
- algoritmul codificat prin funcția respectivă dacă sînt complexe;
- diferite limite impuse datelor de intrare etc.

De asemenea, se pot insera comentarii în corpul funcției unde se consideră că sînt necesare unele explicații.

Comentariile trebuie să fie exprimări clare care să nu conducă la ambiguități și să nu conțină afirmații eronate.

## 1.6. Constante

O constantă are un *tip* și o *valoare*. Atît tipul, cît și valoarea, sînt determinate de caracterele care intră în compunerea constantei. Valoarea unei constante nu poate fi schimbată în timpul execuției programului în care a fost utilizată.

### 1.6.1. Constante întregi

Constantele întregi pot fi scrise în sistemul de numerație cu bază 8, 10

sau 16.

O *constantă zecimală întreagă* este un șir de cifre zecimale care are prima cifră diferită de zero. Constantele zecimale se reprezintă prin complement față de doi pe 16 biți sau pe 32 de biți dacă nu încap pe 16 biți.

Constantele întregi reprezentate pe 16 biți sînt de tip *int*, iar cele reprezentate pe 32 biți sînt de tip *long*.

În cazul în care noi dorim să reprezentăm o constantă zecimală pe 32 de biți, chiar dacă ea se poate reprezenta pe 16 biți, constanta respectivă trebuie să o terminăm prin *L* sau *l*.

**Exemple:**

Reprezentare externă	Reprezentare internă în binar
12345	0011000000111001
123456789	00000111010110111100110100010101
12345L	00000000000000000011000000111001

O constantă zecimală are tipul *unsigned* dacă se termină prin litera *U* sau *u*. O astfel de constantă se reprezintă pe 16 biți dacă nu este mai mare decît 65535 și pe 32 de biți în caz contrar. În cazul în care dorim ca o constantă întreagă fără semn mai mică decît 65536 să se reprezinte pe 32 de biți, constanta respectivă se va termina prin una din următoarele succesiuni de litere:

ul  
lu  
LU  
UL

Constantele întregi fără semn pot fi utilizate pentru a economisi memorie. Astfel, constantele de tip *int* din intervalul [32768,65535] se păstrează pe 32 de biți, în schimb constantele de tip *unsigned* din același interval se reprezintă pe 16 biți.

**Exemple:**

40000u	constantă întreagă de tip <i>unsigned</i> reprezentată pe 16 biți
40000U	idem

4294967295	constantă întreagă de tip <i>unsigned</i> reprezentată pe 32 de biți
40000lu	idem
40000ul	idem
40000LU	idem
40000UL	idem

O *constantă octală* întreagă este o succesiune de cifre octale (0 - 7) precedată de un zero nesemnificativ. O astfel de constantă se păstrează pe 16 biți dacă aceștia îi sunt suficienți și pe 32 de biți în caz contrar. În cazul în care o constantă octală se termină prin *l* sau *L*, ea se păstrează pe 32 de biți chiar dacă sunt suficienți 16 biți pentru reprezentarea ei

Constantele octale sînt de tip *unsigned* dacă se reprezintă pe 16 biți și *unsigned long* dacă se reprezintă pe 32 de biți.

O *constantă hexazecimală* întreagă este o succesiune de cifre hexazecimale precedată de

0x

sau

0X.

În rest, ea are aceleași proprietăți ca și o constantă octală.

Cifrele hexazecimale se obțin extinzînd cifrele zecimale cu literele mici sau mari de la *A* la *F*:

Litera care reprezintă o cifră hexazecimală	Valoare
a sau A	10
b sau B	11
c sau C	12
d sau D	13
e sau E	14
f sau F	15

### Exemple:

Constanta	Tipul constantei	Lungimea reprezentării
123	constanta zecimală de tip int	16 biți
0123	constantă octală de tip unsigned	16 biți
40000	constantă zecimală de tip long	32 biți
040000	constantă octală de tip unsigned	16 biți
0123456	idem	idem
123L	constanta zecimală de tip long	32 biți
0123l	constanta octală de tip unsigned long	32 biți
0x123	constanta hexazecimală de tip unsigned	16 biți
0xa1b2c3	constanta hexazecimală de tip long unsigned	32 biți
0XABCDEF	constanta hexazecimală de tip long unsigned	32 biți

### 1.6.2. Constante flotante

O *constanta flotantă* reprezintă un număr rațional. Ea se compune din

- o parte întreagă care poate fi și vidă;  
o parte fracționară care poate fi și vidă
- un exponent care poate fi și vid.

Evident, nu pot fi vidate toate părțile indicate mai sus. La scrierea unei constante flotante este necesar să fie prezentă fie partea fracționară, fie exponentul precedat de partea întreagă.

*Partea întreagă* este o constantă zecimală.

*Partea fracționară* se compune din caracterul punct după care urmează o succesiune de cifre zecimale. Succesiunea respectivă poate fi vidă numai în cazul în care partea întreagă este prezentă.

*Exponentul* începe cu litera e mică sau mare, după care poate fi prezent un semn opțional (plus sau minus) și un șir de cifre zecimale. Exponentul deținește un factor care exprimă o putere a lui 10.

În exemplele de mai jos și în continuare vom folosi notația \*\* pentru operația de ridicare la putere. Deci  $a$  la puterea  $b$  se va nota prin

$$a^{**}b$$

### Exemple:

Constantă flotantă	Valoare
123.	123
123.7	123,7
.25	0,25
78e4	$78 \cdot 10^{+4}$
.1E-3	$0,1 \cdot 10^{+(-3)}$
123.456e2	12345,6
1234.567e-4	0,1234567

Constantele flotante se reprezintă în dublă precizie (64 biți)

Pentru a reprezenta o constantă flotantă în simplă precizie este suficient să terminăm constanta respectivă prin litera *f* sau *F*

O constantă flotantă terminată prin *l* sau *L* se reprezintă pe 80 de biți și are tipul *long double*.

### 1.6.3. Constantă caracter

Prelucrarea datelor cu ajutorul calculatoarelor are în vedere, printre altele, posibilitatea lucrului pe caractere. În acest scop, caracterele se codifica folosind coduri numerice. Cele mai utilizate coduri sînt codurile EBCDIC (Extended Binary Coded Decimal Interchange Code) și ASCII (American Standard Code for Information Interchange)

La calculatoarele compatibile IBM PC se utilizează codul ASCII. Caracterele acestui cod le împărțim în

*caractere negrafice* - Codul lor este în intervalul {0,31} la care se adaugă și codul 127 (caracterul DEL).

*spațiu* - **Are codul 32.**

*caractere grafice* - Codul lor este în intervalul {33,126}

Caracterele grafice împreună cu spațiul formează setul de caractere *imprimabile*.

O constantă caracter are ca valoare codul ASCII al caracterului pe care-l reprezintă. Ea are tipul *int*.

O constantă caracter corespunzătoare unui caracter imprimabil se reprezintă prin caracterul respectiv inclus între caractere apostrof

### Exemple:

Constantă caracter	Valoare
'a'	97
'A'	65
'0'	48
''	32
'\b'	42

O excepție de la regula de mai sus o reprezintă caracterele apostrof și bara oblică inversă (*backslash*). Astfel, caracterul *backslash* se reprezintă prin două caractere *backslash* incluse între caractere apostrof:

'\\'

La reprezentarea caracterului apostrof se utilizează caracterul *backslash* urmat de un caracter apostrof:

'\'

De aceea, constanta caracter apostrof se poate reprezenta prin:

'\"'

Caracterul *backslash* se poate utiliza pentru a defini constante caracter și pentru caractere negrafice. Așa de exemplu, pentru a defini constanta caracter corespunzătoare tabulatorului se folosește notația cu *backslash*:

'\t'

Deci

'\r'

definește constanta caracter tabulator orizontal. Ea are valoarea 9.

În mod analog, constanta caracter rînd nou (*newline*) se reprezintă prin:

'\n'

Ea are valoarea 10.

Se obișnuiește să se spună că *backslash* introduce o secvență *escape*.

Mai jos se indică reprezentarea unor constante caracter prin secvențe *escape*.



Constantă caracter	Codul ASCII	Denumirea caracterului	Utilizare
'\a'	7	BEL	activare sunet
'\b'	8	BS	revenire cu un spațiu (Backspace)
'\t'	9	HT	tabulator orizontal
'\n'	10	LF	rînd nou (Line Feed)
'\v'	11	VT	tabulator vertical
'\f'	12	FF	salt de pagină la imprimantă (Form Feed)
'\r'	13	CR	retur de car - poziționează cursorul în coloana 1 din rîndul curent (Carriage Return)

La aceste notații adăugăm și utilizările secvenței *escape* pentru a reprezenta caracterele backslash, apostrof și ghilimele.

Constantă caracter	Codul ASCII	Denumirea caracterului
'\"'	34	ghilimele
'\''	39	apostrof
'\\'	92	backslash

Secvența *escape* poate fi folosită pentru a defini constante caracter pentru orice caracter al codului ASCII. Aceasta se realizează folosind codul caracterului respectiv. Astfel, construcția:

'\ddd'

unde:

- d* - Este o cifră octală.
- Reprezintă constanta caracter corespunzătoare codului ASCII egal cu valoarea întregului octal ddd.

**Exemple:**

- '\a' și '\7' - Reprezintă aceeași constantă caracter corespunzătoare caracterului BEL.
- '\b' și '\10' - Reprezintă constanta caracter corespunzătoare caracterului backspace (8 din sistemul zecimal se reprezintă prin numărul 10 în sistemul cu baza opt).

Se observă că în cadrul unei secvențe escape numerele octale nu mai trebuie să fie precedate de un zero ne semnificativ. În acest caz numărul se consideră automat în sistemul cu baza 8.

"\" și \"42" - Reprezintă constanta caracter care corespunde caracterului ghilimele.

La calculatoarele compatibile IBM PC se utilizează un cod *ASCII extins*. Acesta conține 256 de coduri care sînt valori în intervalul [0,255].

Codurile din intervalul [0,127] corespund caracterelor codului ASCII obișnuit, iar cele din intervalul [128,255] corespund unor caractere care au o utilizare specială.

Constantele caracter corespunzătoare caracterelor de cod ASCII din intervalul [128,255] se definesc prin secvențe escape în care se indică codul acestora. De exemplu, constanta caracter

"\377"

corespunde caracterului care are codul ASCII egal cu 255, adică codul ASCII maxim.

Deoarece o constantă caracter este de tip *int*, ea se păstrează pe 16 biți. Pentru constantele caracter mai mari decît 127, se pune problema extensiei semnului. Programatorul poate opta pentru constante caracter cu semn sau fără semn. Pentru a folosi în mod implicit constante caracter fără semn se va alege, în mediul integrat de dezvoltare Turbo C sau Turbo C++ , alternativa *Unsigned Characters* în submeniul *Code Generation* al submeniului *Compile* din meniul *Options*.

### 1.6.4. Șir de caractere

O succesiune de zero sau mai multe caractere incluse între ghilimele formează o constantă șir sau un șir de caractere.

La scrierea caracterelor din compunerea unui șir de caractere se pot utiliza secvențe escape.

**Exemple:**

"Acesta este un șir de caractere"

"Prin secvența escape \" se reprezintă ghilimelele"

"Prin secvența escape \" se reprezintă backslash"

"Apostroful se reprezintă obișnuit"

"s'a"

"alte\tsecvențe\tescape\nîntr-un șir"

Un șir poate fi continuat pe rîndul următor folosind caracterul `backslash`. În acest scop se tastează `backslash` la sfîrșitul rîndului care se continuă, se trece pe rîndul următor (acționînd tasta `Enter`) și se continuă cu tastarea caracterelor șirului respectiv

Caracterul care precede pe `backslash` se va concatena cu primul caracter de pe rîndul următor.

Caracterele unui șir de caractere se pastrează în memorie într-o zonă contiguă, prin codurile lor ASCII. După ultimul caracter al șirului se păstrează caracterul *NUL*, adică valoarea zero. Acesta joacă rolul de marcaj de sfîrșit al oricărui șir de caractere. Din cauza acestui marcaj, trebuie să facem distincție între o constantă caracter care corespunde unui caracter și șirul de caractere care este format din același caracter. Astfel, constanta caracter

`'A'`

se păstrează în memorie într-un octet prin valoarea 65, pe cînd șirul de caractere

`"A"`

ocupă o zonă de doi octeți: în primul octet se păstrează valoarea 65, iar în al doilea caracterul *NUL*, adică valoarea zero.

În concluzie, un șir de caractere se păstrează în memorie într-o succesiune de octeți al cărui număr este egal cu numărul caracterelor șirului respectiv mărit cu 1, deoarece șirul se termină totdeauna prin caracterul *NUL*.

#### Observații:

##### 1. Fie șirul

`"a\1b"`

Acest șir are în compunerea sa:

- caracterul *a* de cod ASCII 97;
- caracterul *SOH* de cod ASCII 1;
- caracterul *b* de cod ASCII 98;
- caracterul *NUL* de cod ASCII 0.

Dacă în locul șirului de mai sus se dorește un șir în care *b* să fie înlocuit prin caracterul 3, atunci scrierea:

`"a\13"`

nu este corectă. Într-adevăr, acest șir are în compunerea sa:

- caracterul a de cod ASCII 97;
- caracterul VT de cod ASCII 11;
- caracterul NUL de cod ASCII 0.

Pentru a reprezenta caracterul 3 în acest caz, va fi nevoie de încă o secvență escape. Secvența escape pentru caracterul 3 este \63, deci șirul de caractere respectiv se scrie astfel:

"a\\1\\63"

## 2. Șirul de caractere

"\\1751"

are în componerea sa:

- caracterul } de cod ASCII 125 (175 în octal);
- caracterul 1 de cod ASCII 49;
- caracterul NUL de cod ASCII 0.

El poate fi scris mai simplu astfel:

"}1"

3. Caracterul NUL nu poate fi utilizat decît la sfîrșitul unui șir de caractere. Aceasta, deoarece un șir de caractere totdeauna se termină la apariția caracterului NUL.

## 1.7. Caractere sau spații albe (white spaces)

În cele ce urmează, prin caracter sau spațiu alb vom înțelege unul din următoarele caractere:

- spațiu (' ');
- tabulator orizontal ('\t');
- caracterul de rînd nou (newline) ('\n').

Menționăm că setul caracterelor albe diferă pentru diferite implementări ale limbajului C. Setul de caractere indicat mai sus este un set minimal comun tuturor implementărilor.

Amintim că un comentariu poate fi inserat într-un program, oriunde este legal să apară un spațiu alb.

## 1.8. Variabile simple, tablouri și structuri

Într-un program utilizăm alături de date constante și date variabile care își schimbă valorile în timpul execuției programului.

Dacă la o dată constantă ne putem referi folosind caracterele din compunerea ei, la o dată variabilă trebuie să ne referim altfel. Cel mai simplu mod este acela de a denumi data respectivă. Numele datei ne permite accesul la valoarea ei, precum și schimbarea valorii dacă este necesar.

În cazul în care o dată nu are legături cu alte date (de exemplu de ordine), vom spune că ea este o dată *izolată*. Numele unei date izolate se spune că reprezintă o *variabilă simplă*.

Unei date izolate îi corespunde un *tip*. În cursul execuției programului se pot schimba valorile unei date variabile dar nu și tipul ei.

Correspondența dintre numele unei date variabile și tipul ei se definește printr-o *declarație*.

Adesea, într-un program este util să considerăm grupe de date. Gruparea datelor se poate face în mai multe moduri.

Un mod simplu de a grupa date este acela de a considera date de *același* tip, în așa fel încât grupa respectivă să formeze o *mulțime ordonată* de elemente la care să ne putem referi folosind *indici*. O astfel de grupă se spune că formează un *tablou*. Unui tablou i se dă un nume. Tipul comun al elementelor unui tablou este și tipul tabloului respectiv. De exemplu, o mulțime ordonată de întregi reprezintă un tablou de tip întreg.

În cazul în care elementele care se grupează într-un tablou sînt ele însele tablouri, vom avea nevoie de mai mulți indici, pentru a ne referi la ele. În cazul în care se utilizează un singur indice pentru a ne referi la elementele tabloului, spunem că tabloul este *unidimensional*. Dacă se folosesc  $n$  indici, se spune că tabloul este  *$n$ -dimensional*.

Exemple simple de tablouri unidimensionale sînt vectorii care au componente de același tip. Așa de exemplu, un vector de componente întregi este un tablou unidimensional de tip întreg.

O matrice de elemente întregi este un exemplu de tablou bidimensional de tip întreg.

Referirea la elementele unui tablou se face printr-o *variabilă cu indici*.

O variabilă cu indici se compune din numele tabloului urmat de valorile indicilor, fiecare indice fiind reprezentat printr-o expresie inclusă între paranteze pătrate.

Valoarea *inferioară* a indicilor este egală cu *zero*. De exemplu, dacă *vect* este un tablou unidimensional de 10 elemente, atunci ne referim la elementele lui cu ajutorul variabilelor cu indici:

<b>vect[0]</b>	<b>primul element</b>
<b>vect[1]</b>	<b>al doilea element</b>
...	
<b>vect[9]</b>	<b>ultimul element</b>

Dacă *mat* este un tablou bidimensional care definește o matrice de 3 linii a 2 coloane fiecare, atunci elementele acestui tablou pot fi referite prin:

<b>mat[0][0]</b>	<b>mat[0][1]</b>	elementele primei linii
<b>mat[1][0]</b>	<b>mat[1][1]</b>	elementele celei de a doua linii
<b>mat[2][0]</b>	<b>mat[2][1]</b>	elementele celei de a treia linii

Un alt mod de a grupa date are în vedere prezența unor *relații* între datele care se grupează. În acest caz datele care se grupează se spune că formează o *structură*. Ele nu neapărat au un același tip.

Prin *structură* înțelegem o mulțime ordonată de elemente "înrudite", ordonare definită de utilizator în vederea simplificării utilizării grupeii de date respective.

Unei structuri i se atașează un *nume*. De asemenea, se atașează câte un *nume* fiecărei componente ale unei structuri, nume care se utilizează la referirea componentelor respective.

Un exemplu simplu de structură este data calendaristică. Aceasta este o grupă care se compune din trei date înrudite:

- zi;
- lună;
- an.

Aceste trei date nu neapărat au toate trei același tip. De exemplu, ziua și anul pot fi de tip *numeric* (int), iar luna poate fi de tip *numeric* dacă ea se reprezintă prin denumire.

Un alt exemplu simplu de structură îi reprezintă numerele complexe. Un număr complex este o mulțime ordonată de două numere, fiecare de tip *double*. Primul număr reprezintă partea reală a numărului complex, iar cel de al doilea, partea lui imaginară.

Structura, ca și variabilele simple și tablourile, corespunde și ea unui tip de dată. Tipul unei structuri nu are nimic comun cu tipurile componentelor sale. Ea este un tip *novu*, diferit de celelalte predefinite în limbaj. Un astfel de tip se spune că este un *tip definit de utilizator* sau mai scurt *tip utilizator*. De

exemplu, data calendaristică definită mai sus reprezintă un tip nou de date, diferit de cele predefinite. De asemenea, numerele complexe reprezentate prin structuri de perechi de numere flotante în dubla precizie, definesc tipul complex.

Tipurile utilizator se definesc printr-o *declarație* care, așa cum vom vedea ulterior, a suferit mai multe modificări. Tot printr-o declarație se stabilește legătura dintre numele unei date structurate și tipul ei.

În general, prin *tip* înțelegem o mulțime de date împreună cu operațiile care pot fi efectuate cu datele respective.

De exemplu, tipul *int* se definește prin mulțimea numerelor întregi din intervalul  $[-32768, 32767]$  reprezentate prin complement față de doi pe 16 biți. Asupra acestor date sunt definite o serie de operații, cum ar fi cele 4 operații aritmetice, operații de comparație etc.

În cazul tipurilor predefinite sunt predefinite atât mulțimea și reprezentarea datelor tipului respectiv, cât și operațiile cu aceste date.

În cazul tipurilor utilizator se definesc de către utilizator, mulțimea și reprezentarea datelor tipului respectiv prin intermediul construcției *struct*. Utilizatorul definește operațiile cu aceste date prin intermediul unor funcții. De exemplu, în cazul tipului *complex* se definește reprezentarea numerelor complexe printr-o construcție *struct*. Operațiile cu numerele complexe se pot realiza apelând funcții corespunzătoare.

Tipurile utilizator pot fi definite ca mai sus prin intermediul facilităților existente în limbajul C. Un neajuns al tipurilor utilizator definite în limbajul C este faptul că nu se stabilește nici o legătură între reprezentarea tipului și funcțiile care definesc operațiile cu datele respective. Acest neajuns a fost înlăturat în C++ prin introducerea noțiunii de *clasă*. Tipurile definite prin intermediul clasei se numesc *tipuri obiect* de date.

În cazul unui tip abstract de date se asigură legătura dintre reprezentarea datelor și funcțiile care definesc operațiile asupra lor. Ca rezultat al acestei legături se obține protecția datelor utilizatorului neavând acces direct la date decât numai prin intermediul funcțiilor definite în acest scop.

Construcția *struct*, precum și definirea claselor se vor prezenta mai târziu.

## 1.9. Declarația de variabilă simplă

În limbajul C nu există declarații implicite. Aceasta înseamnă că orice variabilă înainte de a fi utilizată trebuie declarată.

Declarația unei variabile simple stabilește legătura dintre numele variabilei și tipul valorilor pe care le poate avea variabila respectivă.

În cea mai simplă formă, o declarație de variabilă simplă are formatul:

*tip lista\_de\_nume;*

În calitate de *tip* putem folosi cuvintele cheie ale tipurilor predefinite.

*Lista\_de\_nume* se compune dintr-un nume de variabilă simplă sau mai multe separate prin virgule.

**Exemple:**

1. `int i,j;`

Variabilele *i* și *j* sînt variabile simple de tip *int*. Compilatorul alocă pentru fiecare o zonă de memorie de 16 biți.

Prin *i* ne referim la valoarea conținută în zona de memorie alocată variabilei *i*. De asemenea, tot prin intermediul lui *i* putem atribui sau modifica valoarea din zona de memorie alocată variabilei *i*.

2. `char c;`

Variabila *c* este o variabilă simplă de tip *char*. Ei i se alocă o zonă de memorie de un octet (8 biți).

În această zonă putem păstra un caracter al codului ASCII extins, prin codul lui. Prin *c* ne putem referi la caracterul respectiv.

3. `long double x;`

Variabila *x* este o variabilă simplă de tip *long double*. Ei i se alocă o zonă de memorie de 10 octeți în care se păstrează o valoare flotantă.

## 1.10. Declarația de tablou

Un tablou, ca orice variabilă simplă, trebuie declarat înainte de a fi utilizat. Declarația de tablou, în forma cea mai simplă, conține *tipul comun* elementelor sale, *numele* tabloului și *limitele superioare* pentru fiecare indice, incluse între paranteze pătrate:

*tip nume[lim1][lim2]...[limn];*

unde:

*tip* - Este un cuvînt cheie pentru tipurile predefinite.



*limi* - Este limita superioară a indicelui al i-lea; aceasta înseamnă că indicele al i-lea poate avea valorile:  
 $0, 1, 2, \dots, \text{limi}-1$

Limitele *limi* ( $i=1, 2, \dots, n$ ) sînt *expresii constante*. Prin expresie constantă înțelegem o expresie care poate fi evaluată la compilare în momentul întâlnirii ei de către compilator.

În paragraful precedent am văzut că la elementele unui tablou ne putem referi folosind variabile cu indici.

În limbajul C, numele unui tablou este un simbol care are ca valoare adresa primului său element.

La întâlnirea unei declarații de tablou, compilatorul alocă o zonă de memorie necesară pentru a păstra valorile elementelor sale. Numele tabloului respectiv poate fi utilizat în diferite expresii și valoarea lui este chiar adresa de început a zonei de memorie care i-a fost alocată.

#### Exemple:

1. `int vect[10];`

Declarația de față definește tabloul *vect* de 10 elemente și el are tipul *int*. Pentru acest tablou se alocă  $10 \times 2 = 20$  octeți.

*vect* este un simbol a cărui valoare este adresa primului său element, adică adresa lui *vect[0]*. Deci *vect[0]* are ca valoare valoarea primului element al tabloului, iar *vect* are ca valoare adresa acestui element.

2. `char tab[100];`

Tabloul *tab* este un tablou unidimensional de tip *char*, care are 100 de elemente. I se alocă 100 de octeți și *tab* are ca valoare adresa elementului *tab[0]*.

3. `double dmat[10][50];`

Tabloul *dmat* este un tablou bidimensional de tip *double*. El reprezintă o matrice de 10 linii a 50 de coloane fiecare. Compilatorul rezervă pentru acest tablou

$10 \times 50 \times 8 = 4000$  octeți.

La elementele acestui tablou ne referim prin:

`dmat[0][0] dmat[0][1] .. dmat[0][49]`

`dmat[1][0] dmat[1][1].. dmat[1][49]`

...

`dmat[9][0] dmat[9][1].. dmat[9][49]`

*dmat* are ca valoare adresa elementului `dmat[0][0]`.

## 1.11. Apelul și prototipul funcțiilor

Într-un program o funcție poate avea o definiție și unul sau mai multe apeluri.

Am văzut mai sus că o funcție se definește prin antet urmat de corpul ei. Antetul, de obicei, are formatul:

*tip nume(lista declarațiilor parametrilor formali)*

Lista declarațiilor parametrilor formali este fie vidă, fie conține o declarație de parametru formal sau mai multe separate prin virgulă. Menționăm că declarațiile parametrilor formali aflate într-o listă de felul celei de sus, nu se termină prin punct și virgulă ca cele pentru variabile simple și tablouri.

**Exemple:**

1. `int f(int x, double y)`

Funcția *f* are doi parametri:

*x* de tip *int*

și

*y* de tip *double*.

Funcția returnează o valoare de tip *int*.

2. `double df(long a, int b, unsigned c)`

Funcția *df* are trei parametri:

*a* - de tip *long*;

*b* - de tip *int*;

*c* - de tip *unsigned*.

Ea returnează o valoare de tip *double*.

O funcție poate fi apelată folosind o construcție de formă:

*nume(lista\_parametrilor\_efectivi)*

unde.

- nume* - Este numele funcției care se apelează
- lista parametrilor efectivi* - Este fie vidă dacă funcția nu are parametri, fie se compune din unul sau mai mulți *parametri efectivi* separați prin virgule.
- Un parametru efectiv este o expresie
- Parametri efectivi se corespund cu cei formali prin *ordine* și *tip*.

La apel se atribuie parametrilor formali valorile parametrilor efectivi și apoi execuția se continuă cu prima instrucțiune din corpul funcției apelate. La revenirea din funcție se ajunge în funcția din care s-a făcut apelul și execuția continuă cu construcția următoare apelului.

Pentru a apela o funcție putem utiliza construcția de mai sus urmată de caracterul punct și virgulă.

O altă posibilitate este aceea de a folosi construcția de mai sus drept operand al unei expresii. Un astfel de apel este posibil numai pentru funcțiile care returnează o valoare la revenirea din ele. În acest caz valoarea returnată de funcție se folosește la evaluarea expresiei din care s-a făcut apelul.

Un parametru efectiv de la apelul unei funcții poate fi numele unui tablou. În acest caz, în antetul funcției respective parametrul corespunzător îl vom declara ca fiind tablou.

De exemplu, dacă *tab* este un tablou unidimensional declarat ca mai jos.

```
int tab[100];
```

și *tab* se folosește la apelul funcției *f*:

```
f(tab);
```

atunci funcția *f* are antetul:

```
void f(int x[100])
```

Menționăm că, limita superioară 100 poate fi omisă la declararea parametrului formal *x*, dar nu și parantezele pătrate. Deci aceeași funcție poate avea următorul antet:

```
void f(int x[])
```

În cazul parametrilor formali care sînt tablouri cu mai multe dimensiuni, numai limita primului indice poate fi omisă.

### Exemplu:

Fie declarația

```
double mat[4][10].
```

și apelul

```
fct(mat);
```

Antetul funcției *fct* poate fi următorul.

```
void fct(double mat[][10])
```

O funcție poate fi apelată într-un punct al unui fișier sursă dacă în prealabil a fost definită în același fișier sursă.

### Exemplu:

```
void f1(void) // definiția funcției f1
```

```
{
```

```
...
```

```
}
```

```
void f2(void) // definiția funcției f2
```

```
{
```

```
...
```

```
/* se apeleaza functia f1; ea este în prealabil definită */
```

```
f1();
```

```
...
```

```
}
```

Apelurile funcției nu pot fi precedate totdeauna de definiția ei. În astfel de cazuri definiția funcției apelate este înlocuită printr-un așa numit *prototip* al ei.

Prototipul unei funcții are un format asemănător cu antetul ei. Acesta reprezintă o informație pentru compilator cu privire la:

- tipul valorii returnate de funcție;
- existența și tipurile parametrilor funcției

Aceste informații sînt prezente în antetele funcțiilor. De aceea, un prototip al unei funcții poate fi scris ca și antetul funcției respective, după care se pune punct și virgulă. Există și o formă prescurtată pentru prototip și anume aceea în care se omite numele parametrilor:

```
tip nume(lista tipurilor parametrilor formali);
```

### Exemple:

1. `void f(void);`

Prototipul de față indică faptul că *f* este o funcție fără parametri și care nu returnează nici o valoare.

2. `double a(void);`

Funcția *a* nu are parametri. Ea returnează o valoare flotantă în dublă precizie.

3. `void c(int x,long y[],double z);`

Funcția *c* nu returnează nici o valoare. Are trei parametri:

- primul este de tip *int*;
- al doilea este un tablou unidimensional de tip *long*;
- al treilea este de tip *double*.

4. `void c(int,long[],double);`

Acest prototip exprimă același lucru cu cel precedent.

Compilatorul utilizează datele din prototip pentru a verifica tipurile parametrilor de la apel (parametri efectivi). În cazul în care un parametru efectiv are un tip diferit de tipul corespunzător din prototip, compilatorul C convertește automat valoarea parametrului efectiv spre tipul indicat în prototip.

Utilizatorii limbajelor C și C++ pot folosi o serie de funcții aflate în bibliotecile standard ale acestor limbaje. Apelul unei funcții de bibliotecă implică și el prezența prealabilă a prototipului funcției respective în textul sursă. Pentru a simplifica inserarea în textul sursă a prototipurilor funcțiilor de bibliotecă, s-au construit fișiere cu astfel de prototipuri. Ele au extensia *.h* (header). Un astfel de fișier conține prototipuri pentru funcții de bibliotecă "înrudite". De exemplu, fișierul *stdio.h* conține prototipuri pentru funcțiile de bibliotecă utilizate frecvent în operații de intrare/ieșire, fișierul *string.h* conține prototipurile pentru funcțiile utilizate la prelucrarea șirurilor de caractere etc.

Menționăm că mediile integrate de dezvoltare Turbo C și C++ permit utilizatorului să găsească prototipurile funcțiilor de bibliotecă. În acest scop se procedează astfel:

- se tastează numele funcției pentru care se dorește să se găsească

informații;

- se fixează cursorul pe o literă arbitrară a numelui funcției,
- se tastează <CTRL>-F1.

Se obține prototipul funcției, precum și fișierul cu extensia *.h* care îl conține. De asemenea, se afișează și alte informații în legătură cu funcția respectivă, utile pentru apelurile ei.

## 1.12. Preprocesare

Un program sursă C sau C++ poate fi prelucrat înainte de a fi supus compilării. O astfel de prelucrare se numește *preprocesare*. Ea este realizată automat înaintea compilării. Preprocesarea constă, în principiu, în substituții. Preprocesarea asigură:

- includeri de fișiere cu texte sursă;
- definiții și apeluri de macro-uri;
- compilare condiționată

Preprocesarea se realizează prin prelucrarea unor informații specifice care au ca prim caracter caracterul diez (#)

În paragraful de față vom aborda construcția *#include* și parțial construcția *#define* utilizată la substituiri de succesiuni de caractere. Ulterior se vor descrie și alte facilități oferite de preprocesare.

### 1.12.1. Includeri de fișiere cu texte sursă

Un fișier cu text sursă poate fi inclus cu ajutorul construcției *#include*. Această construcție are unul din următoarele formate

***#include "specificator\_de\_fișier"***

sau

***#include <specificator\_de\_fișier>***

unde:

*specificator de fișier* - Depinde de sistemul de operare. El definește un fișier cu text sursă păstrat pe disc. În faza de preprocesare, textul fișierului respectiv se substituie construcției *#include*. În felul acesta textul fișierului respectiv ia parte la compilare împreună cu textul în care a fost inclus.

În cazul sistemului de operare DOS, specificatorul de fișier trebuie să fie

un nume de fișier împreună cu extensia lui (.C pentru compilatorul C, .CPP pentru compilatorul C++, .H pentru fișiere de tip header (fișiere cu prototipuri etc.). De asemenea, în afară de numele și extensia fișierului, specificatorul de fișier poate conține și o "cale", dacă este necesar, pentru localizarea fișierului.

Diferența dintre cele două formate constă în modul de căutare al fișierului de inclus.

Formatul cu parantezele unghiulare <..> se utilizează la includerea fișierelor *standard*, cum sînt cele care conțin prototipuri pentru funcțiile de bibliotecă. Directoarele în care se caută aceste fișiere se definesc în prealabil cu ajutorul submențiului *Directories* al meniului *Options* din mediul integrat de dezvoltare Turbo C sau Turbo C++. Ordinea de căutare în aceste directoare corespunde cu ordinea în care au fost definite aceste directoare. În cazul în care fișierul căutat nu este găsit în aceste directoare, se dă un mesaj de eroare.

În cazul în care se utilizează caracterele *ghilimele*, fișierul se caută în directorul curent sau conform "căii" dacă aceasta este prezentă.

Un fișier standard care se include frecvent este fișierul *stdio.h* care conține prototipurile pentru o serie de funcții ce realizează operații de intrare/ieșire. El se include folosind construcția:

```
#include <stdio.h>
```

**Exemple:**

1. `#include "fis1.cpp"`

Se include textul fișierului *fis1.cpp* aflat în directorul curent

2. `#include "c:\\tc\\sursă\\fis2.c"`

În acest exemplu este prezentă calea fișierului *fis2.c*. Se observă că pentru a descrie calea, caracterul backslash se dublează conform convenției de reprezentare a caracterului backslash într-un șir de caractere.

Un text inserat cu ajutorul construcției `#include` la rîndul său poate conține construcții `#include` pentru alte fișiere.

Construcțiile `#include` se scriu, de obicei, la începutul fișierelor sursă, pentru ca textele inserate să fie valabile în tot fișierul sursă care se compilează.

## 1.12.2. Substituirii de succesiuni de caractere la preprocesare

Construcția `#define` se poate folosi la substituții de succesiuni de caractere. În acest scop se utilizează formatul:

**#define** *nume succesiune\_de\_caractere*

unde:

*nume* - Este precedat și urmat de cel puțin un spațiu.

Folosind această construcție, preprocesarea *substituie* *nume* cu *succesiune\_de\_caractere* peste tot în textul sursă care urmează construcției `#define` respective, exceptând cazul cînd *nume* apare într-un șir de caractere sau într-un comentariu.

Numele definit printr-o construcție `#define` substituindu-se prin secvența de caractere corespunzătoare nu mai este prezent la compilare.

De obicei, un astfel de nume se scrie cu litere mari pentru a scoate în evidență faptul că el este definit printr-o construcție `#define`.

*Succesiune\_de\_caractere* începe cu primul caracter care nu este alb. Ea poate fi continuată pe mai multe linii terminînd rîndul care dorim să se continue cu backslash.

`#define` este folosită frecvent pentru a defini constante. De aceea, uneori un nume definit printr-o construcție `#define` se spune că este o *constantă simbolică*.

O construcție `#define` autorizează substituția pe care o definește din punctul în care ea este scrisă și pînă la sfîrșitul fișierului în care ea este scrisă sau pînă la întîlnirea unei construcții `#undef` care o anulează. Aceasta are formatul:

**#undef** *nume*

La întîlnirea ei, se dezactivează substituirea lui *nume* cu succesiunea de caractere care i-a fost atașată în prealabil printr-o construcție `#define`.

Succesiunea de caractere dintr-o construcție `#define` poate conține nume care au fost definite în prealabil prin alte construcții `#define`.

**Exemple:**

1.

...

**#define** A 100

/\* A se substitue prin 100 începînd din acest punct al fișierului sursa \*/



```

...
#define FACT 20
#define DIMMAX (A*FACT)
...
char tab[DIMMAX];
...
double mat[A][FACT];
...

```

După preprocesare se obțin declarațiile:

```

...
char tab[(100*20)];
...
double mat[100][20];
...

```

*DIMMAX* s-a substituit prin  $(100 \cdot 20)$  și nu prin valoarea 2000 a produsului deoarece preprocesorul nu face calcule ci numai substituții. Din această cauză se recomandă ca expresiile utilizate în construcțiile `#define` să fie incluse în paranteze rotunde.

2.

```

#define A 123
#define B A+120
...
x=3*B; // se substitue prin x=3*123+120;
...

```

3.

```

#define A 123
#define B (A+120)
...
x=3*B; // se substitue prin x=3*(123+120);
...

```

4.

```

...
#define A 100
...
int x[A+1]; // declaratia devine int x[100+1];
...
#undef A
#define A 3.5

```

```
...
double y;
...
y=A; / instructiunea devine y=3.5;
...
```

În exemplele de mai sus *A* este o constantă simbolică. Ea este un nume atribuit unei constante, nume care însă nu mai este prezent în faza de compilare.

*B* definește o expresie constantă, care este evaluată de compilator la întilnirea ei.

Constantele simbolice se utilizează frecvent în locul constantelor obișnuite deoarece ele prezintă următoarele avantaje

- Permite să se atribue nume sugestive unor constante. Este mult mai sugestiv sa folosim constanta simbolică *PI* definită prin:

```
#define PI 3.14159
```

decît valoarea ei.

- Permite realizarea de prescurtări. De exemplu, dacă valoarea 3.14159 se folosește de mai multe ori în program, atunci este mai simplu să folosim numele *PI* atribuit ei.
- Permite înlocuirea simplă a unei constante printr-o alta. Dacă o constantă se folosește de mai multe ori într-un program și ulterior se constata că valoarea ei trebuie schimbată (de exemplu este o constantă flotanta căreia trebuie să-i schimbăm precizia), atunci este mult mai simplu să-i schimbăm valoarea o singură dată în construcția *#define*, decît sa o căutăm peste tot în program și să-i schimbăm valoarea în mod corespunzător

## 2. INTRĂRI/IEȘIRI STANDARD

Prin *intrări/ieșiri* înțelegem un set de operații care permit schimbul de date între un program și un periferic.

În general, operația de introducere a datelor de la un periferic se numește *citire*, iar cea de ieșire pe un periferic *scriere*.

Numim *terminal standard* terminalul de la care s-a lansat programul. De obicei, terminalele standard sînt de tip *display*. În acest caz operația de scriere se mai numește și *afișare*.

Limbajul C nu dispune de instrucțiuni specifice pentru operațiile de intrare/ieșire. Ele pot fi realizate apelînd *funcții* construite special pentru acest scop. Astfel de funcții există în biblioteca limbajului C. Operațiile de intrare/ieșire realizate prin intermediul funcțiilor de bibliotecă le numim *operații de intrare/ieșire standard*.

Utilizatorul poate el însuși să construiască astfel de funcții, în cazul în care nu sînt utilizabile cele standard existente.

Funcțiile de bibliotecă, utilizate mai frecvent pentru realizarea operațiilor de intrare/ieșire folosind terminalul standard sînt:

- pentru intrări: *getch*, *getche*, *gets* și *sscanf*;
- pentru ieșiri: *putch*, *puts* și *printf*.

La acestea mai adăugăm macrourile.

*getchar* - pentru intrări;

și

*putchar* - pentru ieșiri.

Aceste macrouri sînt definite în fișierul *stdio.h* și din această cauză utilizarea lor implică includerea acestui fișier.

### 2.1. Funcțiile *getch* și *getche*

Funcțiile *getch* și *getche* sînt dependente de implementare. Mai jos indicăm utilizările lor sub medile de programare Turbo C și Turbo C++.

Ambele funcții permit citirea direct de la tastatură a unui caracter.

Funcția *getch* citește de la tastatură *fără ecou*, deci caracterul tastat la *display* nu se afișează pe ecranul acestuia. Ea permite citirea caracterelor de

la tastatură atât a celor corespunzătoare codului ASCII, cât și a celor corespunzătoare unor funcții speciale cum ar fi tastele F1, F2 etc. sau combinații de taste speciale.

La citirea unui caracter al codului ASCII, funcția returnează codul ASCII al caracterului respectiv.

În cazul în care se acționează o tastă care nu corespunde unui caracter ASCII, funcția *getch* se apelează de două ori: la primul apel funcția returnează valoarea zero, iar la cel de al doilea apel se returnează o valoare specifică tastei acționate.

Funcția *getche* este analogă cu funcția *getch*, cu singura diferență că ea realizează citirea cu *ecou* a caracterului tastat. Aceasta înseamnă că se afișează automat pe ecranul terminalului caracterul tastat.

Ambele funcții *nu au parametri* și se pot apela ca operanzi în expresii.

La apelarea lor se vizualizează *ferestra utilizator* și se așteaptă tastarea unui caracter. Programul continuă după tastarea caracterului.

Un apel de forma:

```
getch();
```

se utilizează în cazul în care dorim să vizualizăm fereastra utilizator și să blocăm programul pentru a analiza conținutul curent al ecranului. Pentru a debloca programul se acționează o tastă corespunzătoare unui caracter al codului ASCII. Caracterul respectiv se citește fără *ecou* și apoi se continuă execuția programului.

Funcțiile *getch* și *getche* au prototipurile în fișierul *conio.h*, deci utilizarea lor implică includerea acestui fișier.

## 2.2. Funcția *putch*

Funcția *putch* afișează un caracter pe ecranul terminalului standard. Ea are un parametru care determină imaginea afișată la terminal.

Funcția *putch* poate fi apelată astfel:

```
putch(expresie);
```

Prin acest apel se afișează imaginea definită de valoarea parametrului *expresie*. Valoarea parametrului se interpretează ca fiind codul ASCII al caracterului care se afișează.

Dacă valoarea expresiei se află în intervalul [32,126], atunci se afișează un caracter imprimabil al codului ASCII. Dacă valoarea respectivă este în

afara acestui interval, atunci se afișează diferite imagini care pot fi folosite în diverse scopuri, cum poate fi de exemplu trasarea de chenare.

Funcția *putch* afișează caractere colorate în conformitate cu culoarea curentă setată în modul *text* de funcționare al ecranului.

La revenirea din funcția *putch* se returnează codul imaginii afișate (valoarea parametrului de la apel).

Prototipul funcției *putch* se află în fișierul *conio.h*.

#### Exerciții:

- 2.1 Să se scrie un program care citește un caracter imprimabil și-l afișează apoi pe ecran.

Caracterul respectiv se citește folosind apelul:

```
getch()
```

La revenirea din *getch()* se returnează codul ASCII al caracterului tastat. Acest cod urmează a fi folosit la apelul funcției *putch* pentru a afișa caracterul tastat. Deci pentru a rezolva problema de față putem folosi construcția:

```
putch(getch());
```

În acest caz parametrul efectiv este o expresie formată dintr-un singur operand, operand care constă din apelul funcției *getch*. Aceasta este posibil deoarece orice funcție care returnează o valoare poate fi apelată ca operand al unei expresii.

Menționăm că funcția *getch* returnează valoarea 13 (CR) la citirea caracterului rezultat din acționarea tastei *Enter*.

#### PROGRAMUL BII1

```
#include <conio.h>
main() /* citește fara ecou un caracter imprimabil ASCII
        și-l afișează pe ecran */
{
    putch(getch());
}
```

#### Observații:

1. Programul este inefectiv dacă se acționează o tastă ce nu corespunde caracterelor ASCII.

2. Caracterul afișat prin `putch` se poate vedea în fereastra utilizator. Aceasta se vizualizează tastînd:

`<Alt>-F5`

Se revine în fereastra utilizator acționînd o tastă oarecare

- 2.2. Să se scrie un program care citește fără ecou un caracter imprimabil ASCII, îl afișează la terminal și apoi trece cursorul pe linia următoare.

Programul de față este asemănător cu cel precedent. În plus se cere trecerea cursorului pe linia următoare. În acest scop se apelează funcția *putch* cu codul caracterului *newline*:

```
putch(10);
```

Menționăm că putem înlocui valoarea 10 prin constanta caracter `'\n'`.

```
putch ('\n');
```

## PROGRAMUL BII2

```
#include <conio.h>
main() /* citește fără ecou un caracter imprimabil ASCII, îl afișează la
        terminal și apoi se trece cursorul pe linia următoare */
{
    putch(getch());
    putch ('\n');
}
```

- 2.3. Programul BII3 realizează același lucru ca și programul BII2, cu deosebirea că înainte de a se termina execuția lui, se afișează fereastra utilizator și se așteaptă acționarea unei taste.

## PROGRAMUL BII3

```
#include <conio.h>
main() /* citește fără ecou un caracter imprimabil ASCII, îl afișează,
        trece cursorul pe linia următoare, afișează fereastra uti-
        lizator și se așteaptă acționarea unei taste; programul se ter-
        mina după acționarea unei taste */
{
    putch(getch());
    putch ('\n');
    getch();
}
```

## 2.3. Macrourele `getchar` și `putchar`

Aceste macroure sînt definite în fișierul *stdio.h*. Ele se apelează la fel ca funcțiile.

Macroul `getchar` permite citirea cu ecou a caracterelor de la terminalul standard. Se pot citi numai caractere ale codului ASCII, nu și caractere corespunzătoare tastelor speciale. Prin intermediul macroului `getchar` caracterele nu se citesc direct de la tastatură. Caracterele tastate la terminal se introduc într-o zonă tampon pînă la acționarea tastei *Enter*. În acest moment, în zona tampon, se introduce caracterul de rînd nou (*newline*) și se continuă execuția lui `getchar`. Se revine din `getchar` returnîndu-se codul ASCII al caracterului curent din zona tampon. La un nou apel al lui `getchar` se revine cu codul ASCII al caracterului următor din zona tampon. La epuizarea tuturor caracterelor din zona tampon, apelul lui `getchar` implică tastarea la terminal a unui nou set de caractere care se reincarcă în zona tampon.

Un astfel de mod de desfășurare a operației de citire implică o anumită organizare a memoriei și accesului la caractere, organizare care conduce la noțiunea de *fișier*.

În general, prin fișier se înțelege o mulțime ordonată de elemente păstrate pe suporturi. Elementele unui fișier se numesc *înregistrări*. În mod frecvent fișierele sînt păstrate pe discuri. Cu toate acestea, este util să se considere organizate în fișiere chiar și datele care se tastează sau se afișează la terminal. În acest caz înregistrarea este un rînd aliniat la terminal sau succesiunea de caractere tastată la terminal și terminată prin acționarea tastei *Enter*.

Fișierele conțin o înregistrare specială care marchează *sfîrșitul de fișier*. Această înregistrare se realizează la tastatură prin secvențe speciale. În cazul limbajelor Turbo C și Turbo C++, sfîrșitul de fișier se obține tastînd:

`<CTRL>-Z`

al cărui ecou este:

`^Z`

Macroul `getchar` returnează valoarea constantei simbolice EOF (*End Of File*) la întîlnirea sfîrșitului de fișier. Aceasta constantă este definită în fișierul *stdio.h*. Valoarea ei depinde de implementare. În cazul mediilor de programare Turbo C și Turbo C++, ea are valoarea -1.

Macroul `getchar` se apelează fără parametri și de obicei este un operand al unei expresii: `getchar()`

Macroul *putchar* afișează un caracter al codului ASCII. El returnează codul caracterului afișat sau -1 la eroare.

Se poate apela prin formatul de mai jos:

**putchar**(*expresie*);

Valoarea expresiei reprezintă codul ASCII al caracterului care se afișează.

#### Observații:

1. Citirea caracterelor prin intermediul zonelor tampon are avantajul că permite corectarea erorilor la tastare. Astfel, operatorul de la terminal poate să modifice o secvență de caractere tastate, înainte de a acționa tasta Enter.
2. Macroul *getchar* returnează valoarea 10 (newline) la citirea caracterului corespunzător acționării tastei Enter
3. Apelul  
    **putchar**(10);  
sau echivalentul său  
    **putchar**("\\n");  
are ca efect trecerea cursorului în coloana unu de pe linia următoare.

#### Exerciții:

- 2.4 Să se scrie un program care citește un caracter folosind macroul *getchar*, îl afișează folosind macroul *putchar*, trece cursorul în coloana unu din linia următoare și apoi afișează fereastra utilizator pînă la acționarea unei taste.

Acest exercițiu este analog cu 2.3. În acest caz se cere schimbarea funcțiilor *getch* și *putch* cu macrourile *getchar* și respectiv *putchar*.

#### PROGRAMUL BII4

```
#include <stdio.h>
#include <conio.h>
```

```
main() /* - citește un caracter folosind getchar;
          - afișează caracterul folosind putchar,
          - trece cursorul la începutul rîndului următor,
          - blochează programul pînă la tastarea unui caracter */
```



```

{
    putchar(getchar());
    putchar('\n');
    getch();
}

```

#### Observație:

Deoarece se folosește macroul *getchar*, operatorul va tasta un caracter ASCII care trebuie să fie urmat de acționarea tastei Enter. Fără acționarea tastei Enter, programul rămâne în așteptare. După acționarea tastei Enter se va reveni din *getchar* cu codul ASCII al primului caracter tastat.

Dacă prima tastă acționată este chiar tasta Enter, atunci programul continuă execuția în mod automat și se revine din *getchar* cu valoarea 10 (valoarea corespunzătoare tastei Enter).

## 2.4. Funcțiile gets și puts

Funcția *gets* poate fi folosită pentru a introduce de la terminalul standard o succesiune de caractere terminată prin acționarea tastei Enter.

Citirea se face cu *ecou*. Se pot citi numai caractere ale codului ASCII.

Funcția are ca parametru *adresa de început a zonei de memorie în care se păstrează caracterele citite*. De obicei, această zonă de memorie este alocată unui tablou unidimensional de tip *char*.

Deoarece numele unui tablou are ca valoare adresa de început a zonei de memorie alocate, rezultă că numele unui tablou poate fi utilizat ca parametru în apelul funcției *gets*. În felul acesta, caracterele citite se vor păstra în tabloul respectiv.

Funcția *gets* returnează adresa de început a zonei de memorie în care s-au păstrat caracterele.

La întâlnirea sfârșitului de fișier (<CTRL>-Z) se returnează valoarea zero. Zero nu reprezintă o adresă posibilă pentru *gets* și de aceea, ea poate fi folosită pentru a semnaliza sfârșitul de fișier. De obicei, valoarea returnată de *gets* nu se testează față de zero, ci față de constanta simbolică *NULL* definită în fișierul *stdio.h*.

Caracterul *newline*, care termină rîndul, nu se păstrează în memorie. În locul lui se păstrează caracterul *NUL* ("0"). În felul acesta, caracterele citite prin *gets* formează un șir de caractere în conformitate cu cerințele limbajelor C și C++ (secvența de caractere se termină prin caracterul *NUL*).

Din cele de mai sus rezultă că dacă *tab* este declarat prin:

```
char tab[255];
```

atunci apelul:

```
gets(tab);
```

păstrează în *tab* succesiunea de caractere tastată la terminalul standard în linia curentă. Totodată, *newline* se înlocuiește cu *NUL*.

Funcția *puts* afișează la terminalul standard un șir de caractere ale codului ASCII. Cursorul, după afișarea șirului respectiv, se trece automat în coloana întâi de pe linia următoare (deci caracterul *NUL* se înlocuiește cu *newline*).

Funcția are ca parametru adresa de început a zonei de memorie care conține caracterele de afișat.

În cazul în care șirul de caractere care se afișează se păstrează într-un tablou unidimensional de tip *char*, drept parametru se poate folosi numele acestui tablou.

Funcția *puts* returnează codul ultimului caracter al șirului de caractere afișat (caracterul care precede pe *NUL*, sau -1 la eroare).

Dacă *tab* are declarația de mai sus și el păstrează un șir de caractere, atunci apelul

```
puts(tab);
```

afișează la terminalul standard șirul respectiv de caractere și apoi trece cursorul în coloana întâi de pe rândul următor.

Funcțiile *gets* și *puts* au prototipurile în fișierul *stdio.h*.

### Exerciții:

- 2.5 Să se scrie un program care citește de la intrarea standard numele și prenumele unei persoane, afișează inițialele persoanei respective pe un rând, fiecare inițială este urmată de un punct.

Numele și prenumele se testează pe două rânduri diferite și apoi se vor afișa inițialele respective. În final se trece cursorul în coloana întâi a rândului următor și se afișează fereastra utilizator.

### PROGRAMUL BII5

```
#include <stdio.h>
#include <conio.h>
```

```

main() /* citește numele și prenumele unei persoane tastate
       pe două linii separate;
       - afișează pe un rând inițialele urmate de punct;
       - afișează fereastra utilizator. */
{
    char nume[255];
    char prenume[255];

    gets(nume);          citește prima linie tastată la terminal
    gets(prenume);       citește linia a doua tastată la terminal
    putchar(nume[0]);     afișează prima inițială
    putchar(' ');         afișează punct după prima inițială
    putchar(prenume[0]);  afișează a doua inițială
    putchar(' ');         punct după a doua inițială
    putchar('\n');        trece în coloana întâi din linia următoare
    getch();              se afișează fereastra utilizator, pentru a termina
                          programul se acționează o tastă oarecare
}

```

- 2.6 Să se modifice programul precedent așa încât după afișarea inițialelor să se afișeze textul:

Pentru a termina acționați o tastă

Acest text se poate afișa cu ajutorul funcției *puts*, pe care o apelăm înainte de apelul lui *getch*. Se poate utiliza următorul apel al funcției *puts*:

```
puts("Pentru a termina acționați o tastă");
```

În acest apel parametrul efectiv este o expresie redusă la un operand care este o constantă șir.

La acest apel compilatorul păstrează caracterele constantei șir într-o zonă de memorie și apoi realizează apelul funcției *puts* folosind adresa de început a zonei respective.

## PROGRAMUL BII6

```

#include <stdio.h>
#include <conio.h>

```

```

main() /* citește numele și prenumele unei persoane tastate pe două
       linii separate;
       - afișează inițialele persoanei urmate de puncte pe un același
       rând;

```

- pe rindul urmator se afiseaza textul:  
Pentru a termina programul actionati o tasta;
- se afiseaza fereastra utilizator. \*/

```
{
    char nume[255];
    char prenume[255];

    gets(nume);
    gets(prenume);
    putchar(nume[0]);
    putchar('.');
    putchar(prenume[0]);
    putchar('.');
    puts("\nPentru a termina programul actionati\
        o tasta");
    getch();
}
```

## 2.5. Funcția printf

Funcția *printf* poate fi folosită pentru a afișa date pe ecranul terminalului standard sub controlul unor formate. Ea poate fi apelată printr-o construcție de forma:

```
printf(control,par1,par2,...,pam);
```

unde:

- control* - Este un șir de caractere care definește textele și formatele datelor care se scriu;
- par1,par2,...,pam* - Sint expresii.
  - Valorile lor se scriu conform specificatorilor de format prezenți în parametrul de control.

Datele gestionate prin *printf* sint supuse unor transformări. Aceasta din cauză că datele au un format extern și unul intern.

Parametrul *control* al funcției *printf* definește aceste transformări ale datelor care se afișează pe ecran, transformări care se mai numesc *conversii*. El conține așa numiți *specificatori de format* care definesc conversiile datelor din format intern în format extern. În afara acestor specificatori de format, parametrul *control* al funcției *printf* poate conține succesiuni de caractere care se afișează ca atare în poziții corespunzătoare.

În concluzie, parametrul *control* are în compunerea sa:

- succesiuni de caractere care se afișează ca atare;
- specificatori de format care definesc conversiile valorilor parametrilor *par1*, *par2*, ..., *parn* din format intern în format extern.

Menționăm că aceste construcții nu sînt totdeauna ambele prezente. De exemplu, dacă dorim să se afișeze un text, atunci parametrul *control* nu conține specificatorii de format, decît numai textul respectiv, iar parametrii *par1*, *par2*, ..., *parn* sînt absenți. În mod analog, parametrul *control* poate să conțină numai specificatori de format.

În mod frecvent, parametrul *control* conține atît specificatori de format, cit și alte caractere.

**Exemplu:**

Apelul funcției *puts* folosit în exercițiul 2.6.:

```
puts("\n Pentru a termina programul actionati\  
o tasta");
```

poate fi înlocuit printr-un apel al funcției *printf* care conține numai parametrul de *control* fără specificatori de format:

```
printf("\n Pentru a termina programul actionati\  
o tasta\n");
```

În cazul apelului funcției *printf* s-a folosit caracterul *newline* de la sfîrșitul șirului de caractere pentru ca după afișarea textului, cursorul să treacă în coloana unu din linia următoare. Acest fapt se realizează automat în cazul funcției *puts* și deci la apelul ei șirul de caractere nu se mai termină cu *newline*.

Specificatorii de format, dacă există, se corespund cu parametrii *par1*, *par2*, ..., *parn*. Astfel, al k-lea specificator de format controlează afișarea valorii parametrului *par<sub>k</sub>*.

Formatul extern al unei date constă dintr-o succesiune de caractere imprimabile. Ele se afișează într-o zonă numită *cîmp*.

Un specificator de format începe cu un caracter procent (%). În continuare, mai pot exista caracterele indicate mai jos:

un caracter minus opțional	- Implicit datele se cadrează în dreapta cîmpului în care ele se scriu. Dacă este prezent caracterul minus, atunci data corespunzătoare lui este cadrată în stînga.
un șir de cifre zecimale opțional	- Care definește dimensiunea minimă a cîmpului în care se afișează caracterele care intră în compunerea formatului extern al datei. În cazul în care data necesită un cîmp de

dimensiune mai mare decât cel precizat, ea se va scrie pe atâtea caractere câte îi sînt necesare. În cazul în care data necesită un cîmp mai mic decât cel definit în specificatorul de format, ea se va scrie în cîmpul respectiv în dreapta sau în stînga, dacă este prezent caracterul minus în specificatorul de format corespunzător ei. De asemenea, în acest caz cîmpul se completează cu caractere nesemnificative, implicit, caracterele nesemnificative sînt spații. Caracterele nesemnificative vor fi zerouri dacă numărul ce indică dimensiunea minimă a cîmpului începe cu un zero nesemnificativ (aici zeroul nesemnificativ nu înseamnă că numărul este în octal, deoarece acest șir de cifre nu reprezintă o constantă întreagă, ci are semnificația amintită mai sus).

*un punct  
opțional, urmat  
de un șir de cifre  
zecimale*

- Șirul de cifre zecimale aflate după punct definește precizia datei care se scrie sub controlul specificatorului respectiv. Dacă punctul este prezent, atunci și precizia este prezentă. În cazul în care data care se scrie este flotantă, precizia definește numărul de zecimale care se scriu. În cazul în care data este un șir de caractere, precizia indică numărul de caractere care se scriu.

*una sau două  
litere*

- Care definește tipul de conversie aplicat datei care se scrie.

Din cele de mai sus, rezultă că un specificator de format începe cu un caracter procent și se termină cu o literă.

Între caracterul procent și litera care sfîrșește un specificator de format mai pot apare și alte caractere a căror interpretare s-a dat mai sus.

În cazul în care după caracterul procent urmează un caracter diferit de cele folosite la definirea specificatorilor de format, caracterul procent respectiv se ignoră și se afișează caracterul care urmează după el. Această regulă permite afișarea însuși a caracterului procent. Astfel, succesiunea

%%

nu reprezintă un specificator de format, primul caracter procent se ignoră, iar cel de al doilea caracter procent se va afișa la terminal.

Funcția *printf* returnează numărul de octeți (caractere) care se afișează la terminal sau -1 la eroare.

Prototipul funcției *printf* se află în fișierul *stdio.h*

În cele ce urmează vom indica principalele conversii realizate prin

specificatorii de format. Ele sînt definite de literele de la sfîrșitul specificatorului de format.

### 2.5.1. Litera c

Permite afișarea unui caracter. Valoarea parametrului corespunzător este interpretată ca fiind codul ASCII al unui caracter și caracterul respectiv se afișează în cîmpul definit de specificator.

### 2.5.2. Litera s

Permite afișarea unui șir de caractere.

Parametrul corespunzător are ca valoare adresa de început a zonei de memorie care conține caracterele șirului de afișat. Se afișează toate caracterele șirului pînă la întîlnirea caracterului NUL.

**Exemple:**

1. Secvența de apeluri:

```
putchar(getchar());
```

se poate realiza folosind funcția *printf* astfel:

```
printf("%c",getchar());
```

2. Secvența de apeluri:

```
putchar(getchar());
```

```
putchar('\n');
```

se poate înlocui prin:

```
printf("%c\n",getchar());
```

3. Apelul lui *printf* pentru a afișa un șir de caractere:

```
printf("abc");
```

se poate scrie folosind specificatorul de format pentru șiruri de caractere:

```
printf("%s","abc");
```

În ambele cazuri, compilatorul păstrează șirul de caractere "abc" într-o zonă specială prevăzută pentru acest fapt și apelează funcția *printf* folosind ca parametru adresa de început a acestei zone de memorie.

4. Fie apelul lui *printf*:

```
printf("%c",getchar());
```

Caracterul citit prin *getchar* se afișează într-un cîmp de 4 caractere, cadrat în dreapta. De exemplu, dacă s-a citit caracterul *A*, atunci apelul de mai sus va afișa:

```
* A*
```

5. Apelul:

```
printf("%-4c",getchar());
```

va afișa caracterul citit într-un cîmp de 4 caractere, cadrat în stînga. De exemplu, dacă se citește caracterul *A*, atunci apelul de mai sus va afișa:

```
*A *
```

6. Fie apelul:

```
printf("%10s","abc");
```

se afișează:

```
*   abc*
```

7. Apelul:

```
printf("%-10s","abc");
```

afișează:

```
*abc  *
```

8. Apelul:

```
printf("%15.10s","limbajul C++");
```

afișează:

```
*   limbajul C*
```

### 2.5.3. Litera *d*

Datele de tip *int* pot fi convertite și afișate în zecimal folosind un specificator de format terminat cu litera *d*. Valorile negative se afișează precedate de semnul minus.



### Exemple:

#### 1. Specificatorul

`%d`

se folosește pentru a afișa, în zecimal, o dată de tip *int*.

#### 2. Apelul:

```
printf("**%10d**",123);
```

afișează:

```
*    123*
```

#### 3. Apelul:

```
printf("**%-10d**",123);
```

afișează:

```
*123   *
```

#### 4. Apelul:

```
printf("**%010d**",123);
```

afișează:

```
*0000000123*
```

### 2.5.4. Litera o

Datele de tip *int* sau *unsigned* pot fi convertite și afișate în octal folosind un specificator de format terminat cu litera *o*.

Fiecare grupă de trei cifre binare se înlocuiește printr-o cifră octală. Gruparea cifrelor se face de la dreapta spre stînga, adică de la ordinele inferioare spre cele superioare.

### Exemplu:

Apelul:

```
printf("**%10o**",123);
```

afișează:

```
*    173*
```

### 2.5.5. Literele x și X

Datele de tip *int* sau *unsigned* pot fi convertite și afișate în hexazecimal folosind un specificator de format terminat cu litera *x* sau *X*. În acest scop, patru cifre binare se înlocuiesc printr-o cifră hexazecimală.

**Exemplu:**

Apelul:

```
printf("**%10x*",123);
```

afișează:

```
*      7b*
```

În cazul în care se folosește litera mare *X*, cifrele hexa peste 9 se afișează cu litere mari.

### 2.5.6. Litera u

Litera *u* este asemănătoare cu litera *d*. În acest caz se realizează conversia unei date binare de tip *unsigned* în zecimal.

### 2.5.7. Litera l

Litera *l* poate precede pe una din literele *d*, *o*, *u*, *X* sau *u*.

În acest caz se fac conversii din tipul *long* sau *long unsigned*. Astfel, specificatorul terminat în *ld* realizează conversia din *long* în zecimal. Specificatorul *lu* realizează conversia din *long unsigned* în zecimal. Specificatorul *lo* face conversie din *long* sau *long unsigned* în octal. Specificatorul *lx* sau *lX* face conversia din *long* sau *long unsigned* în hexazecimal.

### 2.5.8. Litera f

Litera *f* permite conversia datelor de tip *float* sau *double* spre formatele *parte\_întreagă.parte\_fracționară*

sau

*part: \_întreagă*

Partea întreagă este un șir de cifre zecimale care este precedat de semnul minus dacă numărul este negativ.

Numărul zecimalelor se definește de precizia indicată în specificatorul

de format. Dacă ea este absentă, atunci se afișează 6 zecimale. Ultima cifră afișată este rotunjită. Rotunjirea se face prin adaus dacă prima cifră neglijată este cel puțin egală cu 5 și prin lipsă în caz contrar.

**Exemple:**

Valoarea datei	Specificator	Afișare
3.14159265	%5f	3.141593
123.672	%7f	123.672000
3.14159265	%7.2f	3.14
123.672	%10.1f	123.7
-123.672	%10.1f	-123.7
3.14159265	%10.0f	3
123.672	%10.0f	124

## 2.5.9. Literele e și E

Aceste litere permit conversia datelor flotante de tip *float* sau *double* spre formatele:

*parte\_intreagă* *parte\_fracționară* *exponent*

sau

*parte\_intreagă* *exponent*

Partea întreagă este o cifră zecimală care este precedată de semnul minus dacă numărul este negativ.

Numărul zecimalelor este cel definit de precizie dacă aceasta este prezentă în specificatorul de format. În cazul în care precizia este absentă, numărul zecimalelor afișate este egal cu 6. Ultima cifră afișată este rotunjită conform regulii indicate mai sus la descrierea literei *f*.

Exponentul începe cu litera *e* dacă specificatorul de format se termină cu *e* și cu *E* dacă el se termină cu *E*. Urmează un semn plus sau minus dacă acesta este negativ. După semn se află un întreg zecimal de cel puțin 2 cifre.

Exponentul definește o putere a lui zece care înmulțește restul reprezentării numărului pentru a obține valoarea reală a acestuia.

**Exemple:**

Valoarea datei	Specificator	Afișare
3.14159265	%e	3.141593e+00
123.672	%e	1.236720e+02
123.672	%.1E	1.2E+02

0.673  
123.672

%E  
%.0E

6.730000E-01  
1E+02

## 2.5.10. Literele g și G

Specificatorul de format terminat cu litera *g* sau *G* afișează data respectivă fie ca în cazul specificatorului terminat cu *f*, fie ca în cazul specificatorului terminat cu *e* sau *E*. Formatul de afișare cu exponent sau fără, se alege automat în așa fel încât afișarea să ocupe un număr minim de caractere. De asemenea, specificatorii *%g* și *%G* afișează 6 zecimale numai dacă acestea sînt toate semnificative.

Se folosește litera *e* la scrierea exponentului dacă specificatorul se termină cu *g* și *E* pentru specificatorul terminat cu *G*.

### Observație:

Datele de tip *float* au o precizie de 6-7 zecimale. De aceea, pentru datele de acest tip nu se recomandă să se afișeze un număr mai mare de zecimale.

Datele de tip *double* au o precizie de pînă la 15 zecimale și deci, în acest caz, putem afișa pînă la 15-16 zecimale. Afișarea unui număr mai mare de zecimale nu are sens.

## 2.5.11. Litera L

Litera *L* poate precede una din literele *f*, *e*, *E*, *g* și *G*. În acest caz data care se afișează este de tip *long double*. Convențiile de afișare sînt aceleași, adică datele afișate cu un specificator de format terminat cu *Lf* sînt fără exponent, cele cu un specificator de format terminat cu *Le* sau *LE* se afișează cu exponent, iar cele afișate cu un specificator de format cu *Lg* sau *LG* se afișează cu unul din formatele precedente care asigură o afișare cu un număr minim de caractere.

### Exerciții:

- 2.7 Să se scrie un program care afișează pe o linie, între două caractere asterisc, caracterul citit de la terminal prin *getchar*, apoi trece cursorul în coloana întâi de pe linia următoare și afișează ecranul utilizator.

## PROGRAMUL BII7

```
#include <stdio.h>
#include <conio.h>
```

```
main() /* - citește un caracter cu getchar;
```

- afiseaza pe o linie caracterul citit, intre doua caractere asterisc;
- trece cursorul in coloana intii din linia urmatoare;
- afiseaza fereastra utilizator. \*/

```
{
printf("**%c*\n",getchar() );
puts(" Actionati o tasta pentru a continua");
getch();
}
```

- 2.8 Să se modifice programul din exemplul precedent în așa fel încît caracterul citit prin *getchar* să se afișeze între două caractere procent.

### PROGRAMUL BII8

```
#include <stdio.h>
#include <conio.h>
```

```
main() /* - citește un caracter cu getchar;
        - afiseaza pe o linie caracterul citit, intre doua
        caractere procent;
        - trece cursorul in coloana intii din linia urmatoare;
        - afiseaza fereastra utilizator. */
```

```
{
printf("%%%c%%\n",getchar() );
puts(" Actionati o tasta pentru a continua");
getch();
}
```

- 2.9 Programul de mai jos afișează textul "Limbaajul C++", folosind următorii specificații de format:

%s, %15s, %-15s, %10s, %15.10s și %-15.10s

Pentru a pune în evidență cimpul afișat prin specificații respective, vom include de fiecare dată cimpul între caractere asterisc.

### PROGRAMUL BII9

```
#include <stdio.h>
#include <conio.h>

#define V "Limbaajul C++"
```

main() /\* afiseaza, intre caractere asterisc, textul Limbajul C++ folosind diferiti specificatori de format \*/

```
{
printf("**%s*\n", V);
printf("**%15s*\n", V);
printf("**%-15s*\n", V);
printf("**%10s*\n", V);
printf("**%15.10s*\n", V);
printf("**%-15.10s*\n", V);
puts (" Actionati o tasta pentru a continua");
getch();
}
```

Rezultatele execuției programului BII9 sînt:

```
*Limbajul C++
*   Limbajul C++
*Limbajul C++  *
*Limbajul C++
*   Limbajul C*
*Limbajul C    *
Actionati o tasta pentru a continua
```

- 2 10 Să se scrie un program care citește un caracter cu *getchar* și afișează codul său ASCII.

#### PROGRAMUL BII10

```
#include <stdio.h>
#include <conio.h>

main() /* citește un caracter cu getchar si afiseaza codul sau ASCII */
{
printf("%d\n", getchar() );
puts( "Actionati o tasta pentru a continua");
getch();
}
```

- 2 11 Să se scrie un program care citește un caracter care nu corespunde codului ASCII și afișează codul său returnat prin funcția *getch*.

Amintim că pentru a citi caractere care nu corespund codului ASCII, se apelează *getch* de două ori. La primul apel, *getch* returnează valoarea zero

La cel de al doilea apel se returnează un cod specific caracterului citit.

#### PROGRAMUL BII11

```
#include <stdio.h>
#include <conio.h>

main() /* citește un caracter care nu aparține codului ASCII și afișează
       codul caracterului citit */
{
    printf("%d\n", getch()); // afișează 0
    printf("%d\n", getch()); // afișează codul caracterului
    puts ("Actionați o tasta pentru a continua");
    getch();
}
```

- 2.12 Să se scrie un program care afișează constanta 12345 în zecimal, octal și hexazecimal.

#### PROGRAMUL BII12

```
#include <stdio.h>
#include <conio.h>

#define V 12345

main() /* afișează întregul 12345 în zecimal, octal și hexazecimal */
{
    printf("zecimal %d\n", V);
    printf("octal %o\n", V);
    printf("hexazecimal %x\n", V);
    puts ("Actionați o tasta pentru a continua");
    getch();
}
```

- 2.13 Să se scrie un program care afișează constanta 123456789 în zecimal, octal și hexazecimal

#### PROGRAMUL BII13

```
#include <stdio.h>
#include <conio.h>
```

```
#define V 123456789
```

```
main() /* afiseaza constanta 123456789 in zecimal, octal  
si hexazecimal */
```

```
{  
    printf("zecimal %ld\n", V);  
    printf("octal %lo\n", V);  
    printf("hexazecimal %lx\n", V);  
    puts("Actionati o tasta pentru a continua");  
    getch();  
}
```

2.14 Programul BII14 afișează constanta 123 cu diferiți specificații de format.

#### PROGRAMUL BII14

```
#include <stdio.h>  
#include <conio.h>  
#define V 123
```

```
main() /* afiseaza constanta 123 cu diferiti specificații de format */
```

```
{  
    printf("%d*\n", V);  
    printf("%7d*\n", V);  
    printf("%-7d*\n", V);  
    printf("%07d*\n", V);  
    puts("Actionati o tasta pentru a continua");  
    getch();  
}
```

Rezultatele execuției programului sînt:

```
*123*
```

```
*      123*
```

```
*123      *
```

```
*0000123*
```

Actionati o tasta pentru a continua

2.15 Programul BII15 afișează numere neîntregi folosind diferiți specificații de format.



## PROGRAMUL BII15

```
#include <stdio.h>
#include <conio.h>

#define A 47.389
#define X -123.5e20

main() /* afiseaza numere neintregi cu diferiti specificatori de format */
{
    printf("A=%f\n", A);
    printf("A=%3f\n", A);
    printf("A=%2f\n", A);
    printf("A=%1f\n", A);
    printf("A=%0f\n", A);
    printf("X=%e\n", X);
    printf("X=%4e\n", X);
    printf("X=%3e\n", X);
    printf("X=%2e\n", X);
    printf("X=%1e\n", X);
    printf("A=%g X=%G\n", A, X);
    printf("%20f\n%20e\n%20g\n", 1.25, 1.25, 1.25);
    printf("%20f\n%20e\n%20g\n", .25, .25, .25);
    printf("%20f\n%20e\n%20g\n", 123e-1, 123e-1, 123e-1);
    printf("%20f\n%20e\n%20g\n", 12.3e4, 12.3e4,
           12.3e4);
    printf("%30.20Le\n", 356.782143657891213891);
    printf("%40.30Le\n",
           31415926535897932384526434e-251);
}
```

Rezultatele execuției programului sînt:

A=47.389000

A=47.389

A=47.39

A=47.4

A=47

X=-1.235000e+22

X=-1.2350e+22

X=-1.235e+22

X=-1.24e+22

X=-1.2e+22

A=47.389 X=-1.235E+22

```

1.250000
1.250000e+00
1.25
0.250000
2.500000e-01
0.25
12.300000
1.230000e+01
12.3
123000.000000
1.230000e+05
123000
3.56782143657891214000e+02
3.141592653589793240000000000000e+00

```

## 2.6. Funcția scanf

Funcția *scanf* poate fi folosită pentru a introduce date tastate la terminalul standard sub controlul unor formate. Ea poate fi apelată printr-o construcție de forma:

```
scanf(control,par1,par2,...,parn);
```

unde:

- control* - Este un șir de caractere care definește formatele datelor și a eventualelor texte aflate la intrarea de la tastatură
- par1,par2,...,parn* - Sunt adresele zonelor în care se păstrează datele citite după ce au fost convertite din formatele lor externe în formate interne corespunzătoare.

Caracterele albe din compunerea parametrului *control* sunt neglijate. Restul caracterelor, care nu fac parte dintr-un specificator de format, trebuie să existe la intrare în poziții corespunzătoare. Acestea se folosesc în vederea efectuării de controale asupra datelor citite.

De exemplu, dacă o dată care se citește se atribuie variabilei *x*, atunci data respectivă poate fi precedată de textul:

```
x =
```

În acest caz parametrul trebuie să conțină în poziția corespunzătoare

```
x = specificatorul de format al datei care se citește
```

Textele de la intrare pot fi precedate de caractere albe care se neglijează.

Specificatorii de format controlează introducerea datelor care au diferite formate externe reprezentate prin anumite succesiuni de caractere. De asemenea, specificatorul de format definește conversiile din formate externe (ale datelor) în cele interne.

Specificatorii de format sînt asemănători cu cei întâlniți la funcția *printf*. Ei încep cu un caracter procent și se termină cu 1-2 litere. Literele definesc tipul conversiei.

Între procent și litere, într-un specificator de format mai putem utiliza, în ordine:

- un caracter asterisc opțional;
- un șir de cifre opțional, care definește lungimea maximă a cîmpului din care se citește data sub controlul formatului respectiv.

*Cîmpul* controlat de un specificator de format începe cu primul caracter curent care nu este alb și se termină:

- a. fie la caracterul după care urmează un caracter alb,
- b. fie la caracterul după care urmează un caracter care nu corespunde specificatorului de format care controlează acel cîmp,
- c. fie la caracterul prin care se ajunge la lungimea maximă a cîmpului, indicată în specificatorul de format

Condiția c. este absentă dacă în specificatorul de format nu este indicată lungimea maximă a cîmpului.

Menționăm ca această regulă nu se aplică la citirea de caractere.

În cazul în care specificatorul de format conține caracterul asterisc, data din cîmpul respectiv va fi prezentă la intrare dar ea nu se atribuie nici unei variabile și deci nu-i va corespunde un parametru, spre deosebire de ceilalți specificatori care nu conțin caracterul asterisc.

Funcția *scanf* citește toate cîmpurile care corespund specificatorilor de format, inclusiv eventualele texte prezente în parametrul *control*.

În cazul unei erori, citirea se întrerupe în locul în care s-a întâlnit eroarea. Eroarea poate proveni:

- din necorespondența textului curent din parametrul de control cu cel din fișierul de intrare;
- din neconcordanța dintre data din cîmp și specificatorul de format sub controlul căruia se face citirea.

Apariția unei erori poate fi pusă ușor în evidență deoarece *scanf*

returnează, la revenirea din ea, numărul cimpurilor citite corect. În felul acesta, valoarea returnată de *scanf* poate fi testată și se poate stabili câte cimpuri au fost citite la fiecare apel a lui *scanf*.

Funcția *scanf* citește date din zona tampon atașată tastaturii, la fel ca și *getchar*. De aceea, datele se citesc efectiv după ce s-a acționat tasta *Enter*.

La întâlnirea sfârșitului de fișier (<CTRL>-Z la mednle Turbo C și Turbo C++ ) se returnează valoarea *EOF*. Se recomandă ca sfârșitul de fișier să fie precedat de un caracter alb pentru a nu pierde data tastată înainte de sfârșitul de fișier. De asemenea, după sfârșitul de fișier se acționează tasta *Enter* pentru a termina înregistrarea curentă

Funcția *scanf*, ca și funcția *printf* are prototipul în fișierul *stdio.h*.

Așa cum s-a arătat mai sus, parametrii *par1*, *par2*, ..., *parn* sînt adresele zonelor de memorie în care se păstrează datele citite de la tastatură în urma conversiilor în format intern. Adresa unei zone de memorie se exprimă adesea folosind operatorul unar *&*.

Așa de exemplu, dacă *i* este o variabilă simplă oarecare, atunci

*&i*

reprezintă adresa zonei de memorie alocată variabilei *i*.

În general, dacă *nume* este numele unei variabile simple, atunci adresa de început a zonei de memorie alocată variabilei *nume* se exprimă cu ajutorul expresiei:

*&nume*

Amintim că dacă *nume* este numele unui tablou, atunci nu vom mai folosi operatorul *&*, deoarece în acest caz *nume* are ca valoare chiar adresa de început a zonei de memorie alocată tabloului respectiv.

## 2.6.1. Litera c

Specificatorul de format

*%c*

se folosește pentru a citi, cu ajutorul lui *scanf*, caracterul curent din zona tampon corespunzătoare datelor introduse de la tastatură.

În acest caz nu se face avans pînă la primul caracter care nu este alb, ci pur și simplu se citește caracterul curent (unul singur). Codul său ASCII se păstrează în zona de memorie definită de parametrul efectiv corespunzător.

### Exemplu:

Fie declarația:

```
char car;
```

Apelul:

```
scanf("%c",&car);
```

citește caracterul curent din zona tampon și păstrează codul ASCII al acestuia în zona de memorie alocată variabilei `car`.

Apelind în continuare funcția *printf*:

```
printf("%c",car);
```

se va afișa, la terminalul standard, caracterul citit prin apelul de mai sus a lui *scanf*.

## 2.6.2. Litera s

Specificatorii de format terminați prin *s* se folosesc pentru a citi șiruri de caractere.

La citirea cu un astfel de specificator de format, data se consideră ca făcând parte dintr-un cîmp care începe cu un caracter care nu este alb și care se termină fie la caracterul după care urmează un caracter alb, fie la caracterul prin care se ajunge la lungimea maximă de caractere indicată în specificatorul de format.

Caracterele citite se pot păstra într-o zonă de memorie organizată ca un tablou de caractere. În acest caz parametrul corespunzător specificatorului de format poate fi chiar numele acestui tablou.

Menționăm că după ultimul caracter citit sub controlul unui astfel de specificator de format se păstrează în mod automat caracterul NUL ('\0').

### Exemple:

#### 1. Fie textul

Limbajul C++

tastat pe un același rînd la terminalul standard.

Dacă *tab1* și *tab2* sînt declarate ca mai jos:

```
char tab1[10];
```

```
char tab2[4];
```

atunci textul de mai sus poate fi citit folosind apelurile:

```
scanf("%s", tab1); // se citește textul. Limbajul
                  / timpul începe cu litera L și se termină cu l
                  // deoarece după l urmează spațiu

scanf("%s", tab2); // se citește textul: C++
                  / timpul începe cu litera C și se termină cu cel
                  / de al doilea caracter + deoarece după el
                  // urmează newline
```

Același lucru se poate realiza folosind un singur apel al lui *scanf* cu doi specificați de format:

```
scanf("%s %s", tab1, tab2);
```

Menționăm că spațiul dintre specificați de format nu este esențial, el se neglijează. Deci același apel poate fi scris și sub forma

```
scanf("%s%s", tab1, tab2);
```

Datele citite prin aceste apeluri se păstrează în memorie prin codurile lor ASCII astfel:

```
tab1[0] -> 'L', tab1[1] -> 'l', tab1[2] -> 'm',
tab1[3] -> 'b', tab1[4] -> 'a', tab1[5] -> 'j',
tab1[6] -> 'u', tab1[7] -> 'l', tab1[8] -> '\0',
tab2[0] -> 'C', tab2[1] -> '+', tab2[2] -> '+',
tab2[3] -> '\0'.
```

## 2. Fie declarațiile:

```
char tab1[10],
```

```
char tab2[10];
```

și apelul:

```
scanf("%2s%9s", tab1, tab2);
```

Presupunem că la terminal s-a tastat cuvântul *recunoscut* precedat și urmat de cel puțin un caracter alb.

În acest caz funcția *scanf* realizează următoarele

Primul specificator de format asigură citirea cuvântului  
ne

Aceasta deoarece timpul are lungimea maximă de 2 caractere. El începe cu litera "n" (primul caracter care nu este alb) și se termină cu litera "e" la care s-a ajuns să se citească două caractere, cit s-a specificat în specificatorul de format.

- Al doilea specificator de format asigură citirea în continuare. În acest caz cimpul începe cu litera "c" și se termină cu litera "t" deoarece după *t* urmează un caracter alb, iar numărul total al literelor cuvintului *cunoscut* este 8, care este mai mic decât lungimea maximă admisă: 9.

În concluzie, după apelul de mai sus, vom avea:

```
tab1[0]->'n'
tab1[1]->'e'
tab1[2]->'0'
tab2[0]->'c'
tab2[1]->'u'
...
tab2[7]->'r'
tab2[8]->'0'
```

### 3. Specificatorul

`%ls`

permite citirea unui singur caracter. O primă deosebire dintre acesta și specificatorul `%c` constă în aceea că, în timp ce `%ls` citește primul caracter care nu este alb, specificatorul `%c` citește caracterul curent, indiferent dacă acesta este alb sau nu. Deci cei doi specificatori citesc același lucru numai dacă, caracterul curent nu este alb.

O altă deosebire dintre cei doi specificatori constă în aceea că, specificatorul `%ls` implică memorarea caracterului *NUL* după caracterul citit, ceea ce nu are loc în cazul specificatorului `%c`. De aceea, secvența

```
char car;
...
scanf("%c",&car);
```

se modifică dacă folosim specificatorul `%ls`, astfel:

```
char car[2];
...
scanf("%ls",car);
```

### 2.6.3. Litera d

Specificatorii terminați prin litera *d* permit citirea întregilor zecimale și conversia lor spre tipul *int*.

Numărul poate fi precedat de un semn. El trebuie să aparțină intervalului [-32768,32767]. În caz contrar, rezultatul va fi imprevizibil.

Cimpul din care se citește întregul se definește conform regulei generale indicate mai sus.

**Exemple:**

**1. Fie declarația**

```
int n;  
și apelul  
scanf("%d",&n);
```

Dacă în cimpul curent din zona tampon de intrare se află întregul

123

precedat și urmat de cel puțin un caracter alb, atunci după acest apel se va păstra, în zona de memorie alocată lui n, valoarea în binar a întregului zecimal 123.

La revenire funcția *scanf* returnează valoarea 1 (un cimp citit corect).

**2. Fie declarațiile**

```
int i1,i2,i3;  
și apelul:  
scanf("%2d %3d %2d",&i1,&i2,&i3),
```

Dacă în zona de intrare se află succesiunea de cifre

1234567

precedată și urmată de caractere albe, atunci după apel variabilele i1, i2 și i3 au respectiv valorile 12, 345 și 67.

Într-adevăr, primul specificator de format asigură citirea cifrelor dintr-un cimp de 2 cifre, al doilea specificator dintr-un cimp de 3 cifre, iar al treilea dintr-un cimp de 2 cifre. La revenire funcția *scanf* returnează valoarea 3 (3 cimpuri citite corect).

**3. Fie declarația**

```
int n;  
și apelul  
scanf("%d",&n);
```

Dacă, la intrare, construcția curentă este



atunci construcția respectivă se consideră eronată, deoarece ea nu este un întreg zecimal. În acest caz *scanf* nu citește nimic și se revine din ea cu valoarea zero. Construcția respectivă va rămâne curentă, deci ea poate fi citită apelînd în continuare funcția *scanf* cu un alt specificator de format (de exemplu *%s*) sau macroul *getchar*.

4. Considerăm declarația și apelul lui *scanf* din exemplul precedent.

Dacă la intrare se află construcția

23i

atunci se va citi întregul 23.

Cimpul începe cu cifra 2 și se sfîrșește cu cifra 3, deoarece după ea urmează o literă, adică un caracter care nu intră în compunerea întregilor zecimali.

La un nou apel al funcției *scanf* se va încerca citirea literei *i*.

După citirea întregului 23 se va reveni din *scanf* cu valoarea 1.

## 2.6.4. Litera o

Litera *o* se utilizează ca și litera *d*, deosebirea constă în aceea că în loc să se citească un întreg zecimal, se va citi un întreg *octal*. În acest caz fiecare cifră octală se înlocuiește cu trei cifre binare care definesc aceeași valoare ca și cifra respectivă.

## 2.6.5. Literele x sau X

Ambele litere se utilizează la fel ca și litera *d*, deosebirea constă în aceea că în loc să se citească un întreg zecimal, se va citi un întreg *hexazecimal*. În acest caz fiecare cifră hexazecimală se înlocuiește cu 4 cifre binare care definesc aceeași valoare ca și cifra respectivă.

Cifrele hexazecimale mai mari decît 9 pot fi reprezentate prin litere mici sau mari.

## 2.6.6. Litera u

Se utilizează pentru a citi întregi zecimali care se păstrează convertiți în binar și au tipul *unsigned*.

## 2.6.7. Litera f

Litera *f* permite citirea numerelor zecimale care pot fi reprezentate atât cu exponent, cât și fără exponent și păstrarea lor în format *floatant simplă precizie* deci date de tip *float*.

## 2.6.8. Litera l

Litera *l* poate precede pe oricare din literele *d*, *o*, *x*, *X* sau *f*.

Dacă *l* precede pe una din literele *d*, *o*, *x* sau *X*, atunci data citită se va converti spre tipul *long* în loc de *int*.

Specificatorul terminat cu *lu* realizează conversia spre *unsigned long*

Specificatorul *lf* realizează conversia numărului citit în *floatantă dublă precizie*, deci spre tipul *double*.

## 2.6.9. Litera L

Litera *L* se folosește la scrierea specificatorilor terminați prin *Lf*. Ea permite conversia numerelor zecimale atât cu exponent, cât și fără exponent, în format *floatant* de tip *long double*.

**Exercițiul:**

2.16 Să se scrie un program care citește un caracter de la terminalul standard cu ajutorul funcției *scanf* și afișează codul ASCII al caracterului respectiv.

### PROGRAMUL BII16

```
#include <stdio.h>
#include <conio.h>

main() /* - citește un caracter folosind funcția scanf;
        - afișează codul ASCII al caracterului respectiv. */
{
    char car;

    scanf("%c", &car);
    printf("%d\n", car);
    puts("Acționați o tasta pentru a continua");
    getch();
}
```

- 2.17 Să se scrie un program care citește numele și prenumele unei persoane separate prin spații albe, apoi afișează prenumele pe un rând și numele pe rândul următor.

#### PROGRAMUL BII17

```
#include <stdio.h>
#include <conio.h>

main() /* - citește numele și prenumele unei persoane separate
        prin spații albe;
        - afișează prenumele pe un rând și numele pe
        rândul următor */
{
    char nume[255];
    char prenume[255];

    scanf("%s %s", nume, prenume);
    puts(prenume);
    puts(nume);
    puts("Acționați o tastă pentru a continua");
    getch();
}
```

- 2.18 Să se scrie un program care citește un întreg zecimal de cel mult 4 cifre și-l afișează în zecimal, octal și hexazecimal.

#### PROGRAMUL BII18

```
#include <stdio.h>

main() /* citește un întreg zecimal de cel mult 4 cifre și-l
        afișează în zecimal, octal și hexazecimal */
{
    int i;

    scanf("%d", &i);
    printf("%d\n", i);
    printf("%o\n", i);
    printf("%x\n", i);
}
```

### Observație:

La execuția acestui program nu se mai afișează fereastra utilizator cu rezultatele. Pentru vizualizarea ei se tastează

<Alt>-F5

Se revine în fereastra de editare acționînd o tastă oarecare.

- 2.19 Să se scrie un program care citește un întreg zecimal de cel mult 9 cifre și-l afișează în zecimal, octal și hexazecimal.

### PROGRAMUL BII19

```
#include <stdio.h>
main() /* citește un întreg zecimal de cel mult 9 cifre și-l
      afișează în zecimal, octal și hexazecimal */
{
    long i;

    scanf("%ld", &i);
    printf("%ld\n%lo\n%lx\n", i, i, i);
}
```

- 2.20 Să se scrie un program care:

- citește o dată calendaristică tastată sub forma:

zzllaa;

afișează data respectivă sub forma:

19aa/ll/zz

unde:

- |    |   |
|----|---|
| zz | - Două cifre ce reprezintă ziua din lună: 01, 02, ...                                       |
| ll | - Două cifre ce reprezintă numărul lunii: 01, 02, ..., 12.                                  |
| aa | - Două cifre ce reprezintă ultimele două cifre ale unui an calendaristic care începe cu 19. |

Programul are la intrare o succesiune de 6 cifre care pot fi precedate și urmate de spații albe. Ele pot fi citite câte două folosind de trei ori specificatorul de format %2d.

### PROGRAMUL BII20

```
#include <stdio.h>
```

```

main() /* - citește o data calendaristică sub forma: zzllaa;
        - o afișează sub forma: 19aa/ll/zz. */
{
    int zi;
    int luna;
    int an;

    scanf("%2d %2d %2d", &zi, &luna, &an);
    printf("19%02d/%02d/%02d\n", an, luna, zi);
}

```

### Observație:

La afișare s-a folosit specificatorul de format:

`%02d`

Acesta afișează cifrele cu un zero ne semnificativ în față. Deci cifrele 1,2,...,9 se afișează ca 01,02,...,09.

2.21 Să se scrie un program care citește datele de mai jos și apoi afișează tipul operației citite pe un rând, iar celelalte date pe rândul următor:

Tip operație	denumire	UM	cod	preț	cantitate
1	REZISTENTA	010KO	123456789	15	1000

### PROGRAMUL BII21

```
#include <stdio.h>
```

```
#define MAXDEN 30
```

```
main() /* citește un rând de date de la intrarea standard și le
        afișează astfel:
```

```
    - tip operație pe un rând;
```

```
    - celelalte date pe rândul următor. */
```

```

{
    char den[MAXDEN+1];
    char tip[2];
    int val;
    char unit[3];
    long cod;
}

```

```

float pret,cantitate;

scanf("%1s%30s %3d %2s %ld %f %f",tip,den,&val,
      unit,&cod,&pret,&cantitate);
printf("%s\n",tip);
printf("%s %03d%s %ld %f %f\n",den,val,unit,
      cod,pret,cantitate);
}

```

#### Observație:

Tipul este un caracter:

- I (intrare);
- E (ieșire);
- T (transfer) etc.

Citirea lui se face folosind specificatorul %1s. Acesta permite eliminarea eventualelor caractere albe care ar putea precede litera ce definește tipul.

## 3. EXPRESII

O *expresie*, în limbajul C, este formată dintr-un *operand* sau mai mulți *legați* prin *operatori*.

### 3.1. Operand

Un operand poate fi:

- o constantă;
- o constantă simbolică;
- numele unei variabile simple;
- numele unui tablou;
- numele unei structuri;
- numele unui tip;
- numele unei funcții;
- referirea la elementul unui tablou (variabilă cu indici);
- referirea la elementul unei structuri;
- apelul unei funcții;
- expresie inclusă în paranteze rotunde.

Unui operand îi corespunde un *tip* și o *valoare*. Dacă tipul operandului este bine precizat la compilare, valoarea operandului se determină fie la compilare, fie la execuție.

**Exemple:**

1. 1234

Este o constantă întreagă zecimală de tip *int*. Reprezintă un operand constant de tip *int*.

2. 0xa1b2

Este o constantă întreagă hexazecimală de tip *unsigned*. Reprezintă un operand constant de tip *unsigned*.

3. Fie declarația

float a1b2;

Numele

`a1b2`

este numele unei variabile simple, deci reprezintă un operand. El are tipul *float*.

#### 4. Fie declarația

```
int tab[10];
```

```
tab[2]
```

Este o referire la elementul al treilea al tabloului *tab*. Reprezintă un operand de tip *int*.

Numele

`tab`

este numele unui tablou și deci el reprezintă un operand. El este de tip "adresă de întregi de tip *int*" (*pointer to int*), adică are ca valoare adresa unei zone de memorie care conține întregi de tip *int*. Se utilizează în mod frecvent ca parametru în apelul funcțiilor.

#### 5. `sum(n,x)`

Este un apel al funcției *sum*. Reprezintă un operand al cărui tip coincide cu tipul valorii returnate de funcția *sum*. Funcția *sum* poate fi apelată ca operand dacă returnează o valoare.

#### 6. `(expresie)`

Este un operand al cărui tip coincide cu tipul expresiei incluse între paranteze.

## 3.2. Operatori

Operatorii pot fi *unari* sau *binari*.

Un operator unar se aplică unui singur operand.

Un operator binar se aplică la doi operanzi. Operatorul binar se aplică la operandul care îl precede imediat și la care îl urmează imediat.

Operatorii limbajului C nu pot avea ca operanzi constante șir (șiruri de caractere). De asemenea, există limite în aplicarea operatorilor la anumiți



operandi.

Mai jos, se descriu operatorii limbajului C. Menționăm că limbajul C++ conține și alți operatori. Ei vor fi descriși ulterior, pe măsură ce se vor defini construcțiile specifice limbajului C++.

Operatorii limbajului C pot fi grupați în mai multe clase: operatori aritmetici, operatori de relație, operatori logici etc.

La scrierea unei expresii se pot folosi operatori din aceeași clasă sau din clase diferite. În principiu, se pot defini expresii complexe în care să se utilizeze operatori din toate clasele. La evaluarea unei astfel de expresii este necesar să se țină seama de *prioritățile* operatorilor care aparțin diferitelor clase de operatori, de *asociativitatea* operatorilor de aceeași prioritate și de *regula conversiilor implicite*.

### 3.2.1. Operatorii aritmetici

Operatorii aritmetici se utilizează la efectuarea calculelor cu date de diferite tipuri predefinite. Aceștia sînt:

- a. operatorii unari + și -;
- b. operatorii binari multiplicativi \* / și %;
- c. operatorii binari aditivi + și -.

Operatorii unari sînt mai prioritari decît cei binari, iar operatorii multiplicativi sînt mai prioritari decît cei aditivi.

Operatorii unari au aceeași prioritate. Ei se asociază de la dreapta spre stînga.

Operatorii multiplicativi au aceeași prioritate. De asemenea, operatorii aditivi au aceeași prioritate.

Operatorii binari se asociază de la stînga la dreapta.

Operatorul unar + nu are nici un efect.

Operatorul unar - are ca efect negativarea valorii operandului pe care-l precede.

Operatorul \* reprezintă operatorul de înmulțire al operandilor la care se aplică.

Operatorul / reprezintă operatorul de împărțire a valorii operandului care îl precede la valoarea operandului care îl urmează.

În cazul în care ambii operandi sînt întregi (*int*, *unsigned* sau *long*), se

realizează o împărțire întreagă, adică rezultatul împărțirii este partea întreagă a citului.

Operatorul `%` se aplică numai la operanzi întregi și are ca rezultat restul împărțirii întregi a valorilor operanzilor săi.

Operatorii binari `+` și `-` reprezintă operațiile obișnuite de adunare și scădere definite în matematică

#### Exemple:

1. `1234` și `+1234`

sînt expresii care au aceeași valoare și tip. Prima se reduce la operandul `1234` care este o constantă de tip *int*.

A doua se compune din operandul `1234` la care se aplică operatorul unar `+`. Cum acesta nu are nici un efect, cele două expresii au aceeași valoare și tip.

2. `-1234`

este o expresie care se compune din operandul `1234` la care se aplică operatorul unar de negativare.

3. `7/3`

Deoarece operanzii sînt întregi, se realizează împărțirea întreagă a lui 7 la 3. Se obține valoarea 2 de tip *int*.

4. `7%3`

Expresia are ca valoare restul împărțirii lui 7 la 3, deci  $7\%3=1$ .

5. `-7%3`

Expresia are ca valoare restul împărțirii lui -7 la 3, deci  $-7\%3=-1$ .

Într-adevăr,

$$-7/3=-2 \text{ și } -7=(-2)*3+(-1),$$

decî restul împărțirii este -1.

#### Exerciții:

- 3.1 Să se scrie un program care citește valoarea lui *x*, calculează valoarea

polinomul  $p(x)=3x^2-8x+7$  și afișează rezultatul. Variabila  $x$  este de tip întreg.

Menționăm că aici și în continuare prin  $a^*b$  notăm  $a$  la puterea  $b$ .

### PROGRAMUL BIII1

```
#include <stdio.h>
main() /* - citește valoarea lui x;
        - calculează valoarea polinomului  $3x^2 - 8x + 7$ ;
        - afișează rezultatul. */
{
    int x;

    printf("tastati valoarea lui x=");
    scanf("%d", &x);
    printf("x=%d\tp(x)=%d\n", x, 3*x*x - 8*x + 7 );
}
```

#### Observație:

Rezultatul calculului este incorect dacă cel puțin un rezultat parțial depășește intervalul valorilor de tip *int* ( $[-32768, 32767]$ ).

În general, dacă operandii unui operator binar au același tip  $t$ , atunci rezultatul aplicării operatorului are tipul  $t$ . În cazul în care rezultatul este în afara intervalului de valori corespunzător tipului  $t$ , rezultatul aplicării operatorului este eronat.

Dacă un operator binar se aplică la operandi de tipuri diferite, se aplică regula conversiilor implicite (vezi mai jos).

- 3.2 Să se scrie un program care citește valoarea lui  $x$ , calculează valoarea polinomului  $f(x)=3,5x^3-9,8x+3,7$  și afișează rezultatul. Calculele se fac în flotantă dublă precizie.

### PROGRAMUL BIII2

```
#include <stdio.h>
main() /* - citește valoarea lui x;
        - calculează și afișează valoarea polinomului
           $3,5x^3 - 9,8x + 3,7$ . */
{
    double x;
```

```
printf("tastati valoarea lui x = ");
scanf("%lf", &x);
printf("x=%g\ta(x)=%g\n", x, 3.5*x*x*x -
      9.8*x + 3.7);
}
```

#### Observație:

Constantele flotante 3.5, 9.8 și 3.7 se consideră de tip *double*.

- 3.3 Să se scrie un program care citește valoarea lui  $x$ , calculează valoarea polinomului  $a(x)=3x^{20}-6x^{16}+8x^9-7x^5+1$  și afișează rezultatul. Calculele se fac în flotantă dublă precizie.

În acest exemplu intervin puteri mari ale lui  $x$ . De aceea, pentru a evita expresii de forma:

$x*x*...*x$

unde  $x$  să se repete de 20 de ori, vom apela funcția *pow* care realizează ridicarea la putere. Ea are prototipul definit în fișierul *math.h* și acesta este:

**double pow(double x, double y);**

La revenire se returnează  $x^{**}y$ , rezultatul fiind de tip *double*.

### PROGRAMUL BIII3

```
#include <stdio.h>
#include <math.h>

main() /* - citește valoarea lui x;
        - calculează și afișează valoarea polinomului:
          a(x)=3x^{20}-6x^{16}+8x^9-7x^5+1. */
{
    double x;

    printf("tastati valoarea lui x = ");
    scanf("%lf",&x);
    printf("x=%g\ta(x)=%g\n",x,
          3.0*pow(x,20.0) - 6.0*pow(x,16.0) +
          8.0*pow(x,9.0) - 7.0*pow(x,5.0)
          + 1.0);
}
```

- 3.4 Să se scrie un program care citește valoarea variabilei  $x$  și a

coeficienților polinomului:

$$q(x) = c_4 \cdot x^4 + c_3 \cdot x^3 + c_2 \cdot x^2 + c_1 \cdot x + c_0,$$

calculează valoarea polinomului  $q(x)$  și afișează rezultatul. Calculele se fac în flotantă dublă precizie.

Calculul valorii unui polinom se poate face folosind funcția de bibliotecă *poly*. Ea are prototipul definit în fișierul *math.h* și este:

**double poly(double x, int n, double c[]);**

Returnează valoarea polinomului de grad  $n$ , pentru valoarea lui  $x$ , coeficienții polinomului fiind  $c[0], c[1], \dots, c[n]$ :  $c[0]$  este termenul liber;  $c[1]$  este coeficientul lui  $x$ ; în general,  $c[i]$  este coeficientul lui  $x$  la puterea  $i$ .

#### PROGRAMUL BIII4

```
#include <stdio.h>
#include <math.h>

main() /* - citește valoarea lui x și coeficienții c[0],c[1],...,c[4] ai
        polinomului
        q(x)=c[4]*x**4+c[3]*x**3+c[2]*x**2+c[1]*x+c[0]
        calculează valoarea polinomului q(x);
        - afișează rezultatul. */
{
    double x,c[5];

    printf("valoarea lui x = ");
    scanf("%lf",&x);
    printf("coeficientii polinomului\n");
    printf("c0 = ");
    scanf("%lf",&c[0]);
    printf("c1 = ");
    scanf("%lf",&c[1]);
    printf("c2 = ");
    scanf("%lf",&c[2]);
    printf("c3 = ");
    scanf("%lf",&c[3]);
    printf("c4 = ");
    scanf("%lf",&c[4]);
    printf("x=%g\tq(x)=%g\n",x,poly(x,4,c));
}
```

3.5 Să se scrie un program care citește cu ecou de la tastatură o literă mică

și o afișează ca literă mare.

Conversile literelor mici în litere mari se realizează simplu în cazul utilizării codului ASCII. Într-adevar, codurile ASCII ale literelor mici se află în intervalul [97,122], iar ale literelor mari în intervalul [65,90]. Aceste coduri corespund literelor în ordine alfabetică:

Litera mică	Codul ASCII	Litera mare	Codul ASCII
a	97	A	65
b	98	B	66
c	99	C	67
...			
z	122	Z	90

Codul ASCII al unei litere mari se obține din codul ASCII al aceleiași litere mici, scăzând valoarea 32. Deci, dacă variabila *lit* are ca valoare codul ASCII al unei litere mici, atunci expresia:

`lit-32`

are ca valoare codul ASCII al aceleiași litere, dar mari.

De obicei, diferența 32 dintre codurile ASCII ale aceleiași litere, se exprimă mai sugestiv prin expresia:

`'a'-'A'`

Această expresie are doi operanzi care sînt constante caracter, deci fiecare are tipul *int*. Înseamnă că tipul expresiei de mai sus este *int*. În acest fel, codul ASCII al literei mari se obține cu ajutorul expresiei:

`lit-('a'-'A')`

care se scrie mai simplu:

`lit-'a'+'A'`

## PROGRAMUL BIII5

```
#include <conio.h>
main() /* citește cu ecou de la tastatură o literă mică și o afișează
        ca literă mare */
{
    putchar(getche() - 'a' + 'A');
}
```

### Observație:

În acest program se presupune că se tastează la terminal o literă mică.

- 3.6 Să se scrie un program care citește cu ecou de la tastatură o literă mare și o afișează ca literă mică.

În acest caz codul literei mari se adună cu valoarea 32, adică cu diferența 'a'-'A'.

### PROGRAMUL BIII6

```
#include <conio.h>
main() /* citește cu ecou de la tastatura o litera mare si o afiseaza
        ca litera mica */
{
    getch(getche() + 'a' - 'A');
}
```

- 3.7 Să se scrie un program care citește un număr ce reprezintă aria unei suprafețe exprimată în jugăre.

Se cere să se exprime aria respectivă în hectare, prăjini pătrate și stinjeni pătrați.

Avem:

1 jugăr = 576 prăjini pătrate = 1600 stinjeni pătrați = 5754,6415 metri pătrați.

### PROGRAMUL BIII7

```
#include <stdio.h>
main() /* - citește un număr ce exprimă aria unei suprafețe în jugăre;
        - afișează aria respectivă în:
        - prăjini pătrate;
        - stinjeni pătrați;
        - hectare. */
{
    double aria;

    scanf("%lf", &aria);
    printf("aria=%g jugare\n", aria);
    printf("aria=%g prajini patrate\n", aria*576);
    printf("aria=%g stinjeni patrati\n", aria*1600);
}
```

```
printf("aria=%g hectare\n", aria*0.57546415);
}
```

### 3.2.2. Regula conversiilor implicite

Această regulă se aplică la evaluarea expresiilor. Ea acționează atunci când un operator binar se aplică la doi operanzi de tipuri diferite.

În cazul în care un operator se aplică la doi operanzi de tipuri diferite, operandul de tip "inferior" se convertește spre tipul "superior" al celuilalt operand și rezultatul este de tipul "superior".

Regula conversiilor implicite are în vedere și niște conversii generale independente de operatori. Ea se aplică în mai mulți pași, în ordinea indicată mai jos.

Înainte de toate se convertesc operanzii de tip *char* și *enum* (acest tip va fi definit mai târziu) în tipul *int*.

Dacă operatorul curent se aplică la operanzi de același tip, atunci se execută operatorul respectiv, iar tipul rezultatului coincide cu tipul comun al operanzilor. Dacă rezultatul aplicării operatorului reprezintă o valoare în afara limitelor tipului respectiv, atunci rezultatul este eronat (are loc o "depășire").

Dacă operatorul binar curent se aplică la operanzi diferiți, atunci se face o conversie înainte de execuția operatorului, conform pașilor de mai jos:

1. Dacă unul din operanzi este de tip *long double*, atunci celălalt operand se convertește spre tipul *long double* și rezultatul aplicării operatorului este de tip *long double*.
2. Altfel, dacă unul din operanzi este de tip *double*, atunci celălalt operand se convertește spre tipul *double* și rezultatul aplicării operatorului este de tip *double*.
3. Altfel, dacă unul din operanzi este de tip *float*, atunci celălalt operand se convertește spre tipul *float* și rezultatul aplicării operatorului este de tip *float*.
4. Altfel, dacă unul din operanzi este de tip *unsigned long*, atunci celălalt operand se convertește spre *unsigned long* și rezultatul aplicării operatorului este de tip *unsigned long*.
5. Altfel, dacă unul din operanzi este de tip *long*, atunci celălalt operand se convertește spre tipul *long* și rezultatul aplicării operatorului este de



tip *long*.

6. Altfel, unul din operanzi trebuie să fie de tip *unsigned*, celălalt de tip *int* și acesta se convertește spre *unsigned*, iar rezultatul aplicării operatorului este de tip *unsigned*.

Aplicind regula de mai sus, la fiecare operator curent, în procesul de evaluare a unei expresii, se determină în final tipul expresiei respective.

**Exemple:**

1. `int c;`

...

`c-'a'+'A'`

Constantele caracter 'a' și 'A' sînt de tip *int*, deci toți operanzii sînt de tip *int*. Nu este necesară nici o conversie pentru evaluarea expresiei.

2. `int i;`

...

`12345*i`

Operanzii sînt de tip *int*, expresia este de tip *int*; rezultatul înmulțirii este eronat dacă este în afara intervalului [-32768,32767].

3. `int i;`

`long double a;`

...

`a*3+i`

Ținînd seama de prioritatea operatorilor, întii se execută înmulțirea, apoi adunarea; expresia se evaluează astfel:

- 3 se convertește spre *long double* (regula 1), se realizează înmulțirea și rezultatul este de tip *long double*;
- la rezultatul înmulțirii se adună *i* după ce, în prealabil, *i* se convertește spre *long double*, deci expresia este de tip *long double*.

**Exerciții:**

- 3.8 Să se scrie un program care citește doi întregi de la intrarea standard și afișează media lor cu o zecimală.

## PROGRAMUL BIII8

```
#include <stdio.h>
main() /* citește doi întregi și afișează media lor aritmetică
        cu o zecimală */
{
    int a,b;

    scanf("%d %d", &a,&b);
    printf("a=%d\tb=%d\tmedia=%.1f\n",a,b,(a+b)/2.0f);
}
```

### Observații:

1. Pentru calculul mediei se folosește expresia:

$$(a+b)/2.0f$$

Ea se evaluează astfel:

- Se calculează suma  $a+b$ ; rezultatul este de tip *int*, deoarece ambii operanzi sînt de tip *int*. Dacă suma nu aparține intervalului  $[-32768, 32767]$ , rezultatul este cronat.
- Rezultatul adunării se convertește spre *float*, deoarece celălalt operand este de tip *float*, apoi se face împărțirea și rezultatul este de tip *float*.

2. Dacă s-ar fi utilizat expresia

$$(a+b)/2.0$$

suma  $a+b$  s-ar fi convertit spre *double* deoarece împărțitorul este de tip *double*; în acest caz rezultatul este de tip *double*.

3. Expresia

$$(a+b)/2$$

realizează împărțirea întreagă dintre operanzii  $a+b$  și 2 care sînt de tip *int*; de aceea, pentru  $a$  și  $b$  de parități diferite, expresia de mai sus nu dă rezultatul corect.

- 3.9 Să se scrie un program care citește măsura unui unghi în grade sexagesimale și afișează valoarea funcției sinus pentru unghiul respectiv.

Măsura unghiului în grade sexagesimale se tastează sub forma:

g m s

unde:

- g - Întreg ce reprezintă gradele.
- m - Întreg ce reprezintă minutele.
- s - Întreg ce reprezintă secunde.

Pentru calculul funcției sinus se apelează funcția *sin* aflată în biblioteca limbajelor C și C++.

Prototipul ei este:

**double sin(double);**

și se află în fișierul *math.h*.

Parametrul funcției este măsura unghiului în radiani pentru care se calculează valoarea funcției *sin*. Conversia măsurii unghiului din grade sexagesimale în radiani se realizează folosind expresia:

$(g+m/60.0+s/3600.0)*PI/180.0$

unde

$PI = 3.14159265358979$

### PROGRAMUL BIII9

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

main() /* citește măsura unui unghi în grade sexagesimale și afișează
        valoarea funcției sin pentru unghiul respectiv. */
{
    int g,m,s;

    scanf("%d %d %d", &g,&m,&s);
    printf("grade:%d\tminute:%d\tsecunde:%d\n",g,m,s);
    printf("sin=%.13f\n",sin((g+m/60.0 +
    s/3600.0)*PI/180.0));
}
```

### 3.2.3. Operatorii de relație

Operatorii de relație sînt 4:

- < - mai mic;
- <= - mai mic sau egal;
- > - mai mare;
- >= - mai mare sau egal.

Ei au aceeași prioritate, care este mai mică decât a operatorilor aditivi.

O expresie de forma

*E1 operator\_de\_relatie E2*

unde:

*E1 și E2* - Sînt expresii, are una din valorile 0 sau 1.

Astfel, expresia are valoarea 1, dacă valorile celor două expresii satisfac relația prin care acestea sînt legate și 0 în caz contrar.

De obicei, se spune despre o expresie relațională că ea are valoarea *adevărat* dacă subexpresile ei satisfac operatorul relațional prin care sînt legate; în caz contrar se spune că expresia are valoarea *fals*.

În limbajele C și C++ nu există valori logice speciale. Valoarea *fals* se reprezintă prin valoarea 0 (zero), iar valoarea *adevărat* printr-o valoare diferită de zero.

**Exerciții:**

3.10 Să se scrie un program care citește două numere, primul este de tip *int*, iar al doilea este de tip *double*. Programul afișează valoarea 2 dacă numărul flotant este mai mare decât cel întreg și 1 în caz contrar.

Presupunem că primul număr se atribuie variabilei *i*, iar cel de al doilea variabilei *f*.

Atunci expresia

*f > i*

are valoarea 1 dacă numărul flotant este mai mare decât cel întreg și 0 în caz contrar. Pentru evaluarea relației de mai sus, valoarea variabilei *i* se convertește spre *double* și abia apoi se execută operatorul de relație. Tipul expresiei este *int*.

Programul de față trebuie să afișeze valoarea acestei expresii mărită cu 1, ceea ce se obține folosind expresia

*(f > i)+1*

Parantezele sînt necesare deoarece operatorul + este mai prioritar decât >. În lipsa lor, se compară valoarea lui *f* cu valoarea sumei *i+1*.

## PROGRAMUL BIII10

```
#include <stdio.h>
main() /* - citește pe i și f;
        - afișează 2 dacă f este mai mare decât i și 1 în caz contrar */
{
    int i;
    double f;

    scanf("%d %lf", &i, &f);
    printf("i=%d\tf=%f\t%d\n", i, f, (f > i) + 1 );
}
```

### 3.2.4. Operatorii de egalitate

Operatorii de egalitate sunt doi:

== - egal;  
!= - diferit.

Ei au aceeași prioritate care este imediat mai mică decât cea a operatorilor de relație.

O expresie de forma:

*E1 operator\_de\_egalitate E2*

unde:

*E1 și E2* - Sînt expresii, are una din valorile 0 sau 1.

Astfel, expresia de mai sus are valoarea 1, dacă valorile expresiilor *E1* și *E2* satisfac operatorul de egalitate prin care sînt legate și zero în caz contrar. În primul caz, se obișnuiește să se spună că expresia de mai sus este *adevărată*, iar în cel de al doilea caz, că ea este *falsă*.

**Exemple:**

```
1.  int a;
    long b;

    Expresia
    b==a*a
```

are valoarea adevărat (1) dacă *b* este egal cu pătratul lui *a*; altfel ea are valoarea fals (0).

```
2.  int x;
```

Expresia

$x \% 4 == 0$

are valoarea adevărat dacă  $x$  este multiplu de 4 și fals în caz contrar.

3.    `int x;`

Expresia

$x \% 100 != 0$

are valoarea adevărat dacă  $x$  nu este multiplu de 100 și zero în caz contrar.

4.

Expresia

`getchar() != EOF`

are valoarea adevărat dacă macroul *getchar* a citit un caracter diferit de sfârșitul de fișier.

5.    `int x;`

`scanf("%d",&x) == 1`

Această expresie are valoarea adevărat dacă *scanf* a citit de la terminalul standard un întreg (*scanf* returnează numărul cimpurilor citite de la terminalul standard). În caz contrar, adică dacă la intrare nu se află un întreg, expresia de mai sus are valoarea fals.

### 3.2.5. Operatorii logici

Aceștia sint:

`!`    - negația logică (operator unar);

`&&`   - și logic;

`||`   - sau logic.

Negația logică are aceeași prioritate ca și ceilalți operatori unari ai limbajului C. De altfel, toți operatorii unari au aceeași prioritate care este imediat mai mare decât cea a operatorilor multiplicativi.

Expresia

`!operand`

are valoarea zero (fals) dacă *operand* are o valoare diferită de zero și

valoarea unu (adevărat) dacă *operand* are valoarea zero.

Operatorul `&&` are prioritatea mai mică decât cea de egalitate. O expresie de forma:

*E1&&E2*

are valoarea 1, dacă expresiile *E1* și *E2* sunt ambele diferite de zero. În rest, expresia de mai sus are valoarea zero.

Operatorul `||` are prioritatea imediat mai mică decât operatorul `&&`  
Expresia

*E1||E2*

are valoarea zero, dacă ambele expresii *E1* și *E2* la care se aplică operatorul `||` au valoarea zero. În rest, expresia de mai sus are valoarea unu.

Operatorii logici binari (`&&` și `||`) se evaluează de la stînga la dreapta. Dacă la evaluarea unei expresii se ajunge într-un punct în care se cunoaște valoarea de adevăr (fals sau adevărat) a întregii expresii, atunci restul expresiei, aflată în dreapta punctului respectiv, nu se mai evaluează.

**Exemple:**

1. `!a&&b`

Dacă *a* are valoarea zero și *b* este diferit de zero, atunci expresia de mai sus are valoarea unu (adevărat).

Într-adevăr, dacă *a* are valoarea zero, atunci `!a` are valoarea unu.

Dacă *b* este diferit de zero, atunci ambii operanzi ai operatorului `&&` sînt diferiți de zero și deci întreaga expresie are valoarea unu.

2. `!a||b`

Această expresie are valoarea zero dacă *a* are valoarea diferită de zero, iar *b* are valoarea zero. În acest caz `!a` are valoarea zero și deci ambii operanzi ai operatorului `||` au valoarea zero. Conform definiției operatorului `||`, rezultă că întreaga expresie are valoarea zero.

3. `!a&&b||!b&&a`

Această expresie are valoarea zero dacă *a* și *b* au ambii valoarea zero sau ambii sînt diferiți de zero. În caz contrar, expresia are valoarea unu. Acest lucru rezultă din tabela de mai jos:

a	b	!a	!b	!a&&b	!b&&a	!a&&b  !b&&a
0	0	1	1	0	0	0
diferit de zero	diferit de zero	0	0	0	0	0
diferit de zero	0	0	1	0	1	1
0	diferit de zero	1	0	1	0	1

Din cele de mai sus rezultă că expresia

**!a&&b || !b&&a**

realizează *sau logic exclusiv* între *a* și *b*.

Pentru  $a=0$  și *b* diferit de zero, expresia de mai sus se evaluează astfel:

1. Se evaluează *!a* și se obține valoarea unu.
2. Se aplică operatorul *și logic* la *!a* și *b* și se obține unu.

Deoarece *!a&&b* este legat de restul expresiei prin operatorul *sau logic*, rezultă că în acest moment se poate afirma că întreaga expresie este adevărată (are valoarea unu). De aceea, în acest caz nu se mai evaluează restul expresiei, adică termenul *!b&&a*.

### Exerciții:

- 3.11 Să se scrie un program care citește un număr și afișează unu dacă numărul respectiv aparține intervalului  $[-1000, 1000]$  și zero în caz contrar.

Dacă notăm cu *a* numărul citit, atunci el trebuie să satisfacă simultan relațiile:

$a \geq -1000$

și

$a \leq 1000$

Acest fapt se poate exprima cu ajutorul expresiei:

$a \geq -1000 \ \&\& \ a \leq 1000$ .



## PROGRAMUL BIII11

```
#include <stdio.h>
main() /* citește pe a și afișează unu dacă numărul citit aparține
        intervalului [-1000,1000] și zero în caz contrar */
{
    double a;

    scanf("%lf",&a);
    printf("a=%g\t%d\n", a, a >= -1000 && a <= 1000);
}
```

3.12 Să se scrie un program care citește un întreg din intervalul [1600,4900], ce reprezintă un an calendaristic, afișează unu dacă anul este bisect și zero în caz contrar sau dacă anul nu aparține intervalului indicat mai sus

Un an, din calendarul gregorian, este bisect dacă este multiplu de patru și nu este multiplu de o sută sau dacă este multiplu de patru sute. Această regulă este valabilă pentru anii care nu sînt anteriori anului 1600

Calendarul gregorian a fost introdus datorită faptului că anul iulian a fost prea lung.

Conform calendarului iulian, un an era bisect, dacă era divizibil cu patru. Din această cauză la 133 de ani se înregistrează o diferență de o zi față de anul tropic. În felul acesta, în anul 1582 s-a constatat o diferență de 10 zile întârziere și s-a hotărît corectarea diferenței respective convenind ca după ziua de joi 4 octombrie 1582 să urmeze ziua de vineri 15 octombrie 1582. De asemenea, s-a stabilit ca durata unui an calendaristic să fie:

$$365 + 1/4 - 1/100 + 1/400 = 365,2425 \text{ zile}$$

În această expresie termenii au semnificația:

- |       |   |
|-------|---|
| 365   | - Numărul de zile ale anului iulian.  |
| 1/4   | - Adăugarea de un sfert de zi, adică de o zi la patru ani, conform calendarului iulian.                                   |
| 1/100 | - Eliminarea unei zile la fiecare o sută de ani. Deci un astfel de an nu mai este bisect, deși el este multiplu de patru. |
| 1/400 | - Adăugarea unei zile la fiecare 400 de ani. În felul acesta, un an multiplu de 400 are o zi în plus, deci este bisect    |

Calendarul gregorian, avînd durata anului egală cu 365, 2425, are o eroare de aproximativ 26 de secunde. Aceasta înseamnă că regula indicată mai sus pentru a stabili anii bisecți (care au 366 de zile) este valabilă pentru

anii urmatori anului 1582 și care nu sint ulterioari anului 4900.

Într-adevăr, eroarea de 26 de secunde dintr-un an gregorian devine o zi abea la aproximativ 3323 de ani, deci după anul 4900 ( $1582 + 3323 = 4905$ ). De aceea, regula de mai sus se poate aplica pentru anii din intervalul [1600,4900].

Dacă notăm cu  $an$  anul calendaristic, atunci el este bisect dacă de exemplu:

$an \% 4 == 0$  (an este multiplu de 4)

și

$an \% 100 != 0$  (an nu este multiplu de o sută).

Deci anul este bisect dacă expresia

(1)  $an \% 4 == 0 \ \&\& \ an \% 100 != 0$

este adevărată.

De asemenea, anul este bisect și în cazul în care expresia

(2)  $an \% 400 == 0$  (an este multiplu de 400)

este adevărată. Deci anul este bisect dacă este adevărată expresia (1) sau (2), adică dacă este adevărată expresia:

$an \% 4 == 0 \ \&\& \ an \% 100 != 0 \ || \ an \% 400 == 0$

## PROGRAMUL BIII12

```
#include <stdio.h>
main() /* citește un întreg care reprezintă un an calendaristic din
        intervalul [1600,4900] și afișează 1 dacă anul este bisect și
        zero dacă nu este sau nu aparține intervalului indicat */
{
    int an;

    scanf("%d", &an);
    printf("an=%d\t%d\n", an,
           an >= 1600 && an <= 4900 && (an \% 4 == 0 &&
           an \% 100 != 0 || an \% 400 == 0));
}
```

3.13 Să se scrie un program care citește un întreg din intervalul [1600,4900] ce reprezintă un an calendaristic și afișează numărul de zile din anul respectiv.

## PROGRAMUL BIII13

```
#include <stdio.h>
main() /* citește un întreg care reprezintă un an calendaristic din
        intervalul [1600,4900] și afișează numărul de zile din
        anul respectiv */
{
    int an;

    scanf("%d", &an);
    printf("anul %d are %d zile\n", an,
           365 + (an%4 == 0 &&
                 an%100 != 0 || an%400 == 0));
}
```

### Observație:

În acest program se consideră că anul tastat la terminalul standard aparține intervalului [1600,4900].

## 3.2.6. Operatorii logici pe biți

Operatorii logici pe biți sînt:

- ~ - complement față de unu (operator unar);
- << - deplasare stînga;
- >> - deplasare dreapta;
- & - și logic pe biți;
- ^ - sau exclusiv logic pe biți;
- | - sau logic pe biți.

Acești operatori se aplică la operanzi de tip întreg. Ei se execută bit cu bit și operandii se extind la 16 biți dacă este necesar.

Complementul față de unu are aceeași prioritate ca ceilalți operatori unari. El schimbă fiecare bit 1 al operandului cu zero și fiecare bit zero al acestuia cu 1.

### Exemple:

1. ~ 1234

1234 se reprezintă în binar astfel:

10011010010

Se extinde la 16 biți:

0000010011010010

apoi se aplică operatorul de complementare față de 1

1111101100101101

sau în octal:

0175455

## 2. $\sim -1234$

Operatorii unari au aceeași prioritate și se asociază de la dreapta la stînga, deci întîi se realizează negativarea și apoi complementarea față de 1.

Numarul 1234 se reprezintă în binar,

10011010010

se negativează:

1111101100101110

apoi se completează față de 1:

10011010001

sau în octal:

02321

## 3. $- \sim 1234$

În acest caz, întîi se determină complementul față de 1, apoi se negativează numărul.

1234 în binar este:

10011010010

se completează față de 1:

1111101100101101

se negativează:

10011010011

sau în octal:

02323

Operatorii de deplasare sînt operatori binari. Ei au aceeași prioritate,

care este imediat mai mică decât a operatorilor aditivi și imediat mai mare decât prioritatea operatorilor relaționali.

Operatorul  $<<$  realizează o deplasare la stînga a valorii primului său operand cu un număr de poziții binare egal cu valoarea celui de-al doilea operand al său. Această operație este echivalentă cu înmulțirea cu puteri ale lui 2.

În mod analog, operatorul  $>>$  realizează o deplasare la dreapta a valorii primului său operand cu un număr de poziții binare egal cu valoarea celui de-al doilea operand al său. Această operație este echivalentă cu o împărțire cu puteri ale lui 2.

**Exemple:**

1. `int a;`

`a<<3`

are aceeași valoare ca și

`a*8`

adică se înmulțește a cu 2 la puterea 3

2. `int a;`

`a>>4`

are aceeași valoare ca și

`a/16`

adică a se împarte la 2 la puterea 4.

3. `~0<<3`

Această expresie se evaluează astfel:

a. Se extinde zero pe 16 biți:

0000000000000000

b. Se aplică operatorul de complementare față de unu:

1111111111111111

c. Se fac 3 deplasări spre stînga:

1111111111111000

sau în octal:

177770

#### 4. $\sim(\sim 0 < 3)$

Rezultatul din exemplul precedent se completează față de unu:

000000000000111

Operatorul *și logic pe biți* se execută bit cu bit, conform tabelii de mai jos:

$1 \& 1 = 1$

$1 \& 0 = 0$

$0 \& 1 = 0$

$0 \& 0 = 0$

Operatorul  $\&$  are prioritatea imediat mai mică decât operatorii de egalitate. Se utilizează la anulări de biți.

**Exemple:**

#### 1. `int a;`

`a&0377`

are ca valoare, valoarea octetului mai puțin semnificativ al valorii variabilei *a*.

Într-adevăr, se extinde constanta octală 0377 la 16 biți:

0000000011111111

apoi se face *și logic pe biți* între această constantă și valoarea variabilei *a*. Se păstrează ultimii 8 biți ai lui *a*, primii 8 biți anulându-se.

#### 2. `int a;`

`a&0177400`

are ca valoare, valoarea lui *a* după ce s-au anulat ultimii 8 biți ai săi. Aceasta rezultă din faptul că, constanta octală 0177400 are în binar valoarea:

1111111100000000

#### 3. `int a;`

a&0177776

are ca valoare cel mai mare număr par care nu-l depășește pe  $a$ .

Diferența dintre operatorul *și logic* (&&) și operatorul *și logic pe biți* (&) constă în aceea că primul se realizează global, față de al doilea care se realizează pe biți.

Așa de exemplu, dacă  $x=2$  și  $y=1$ , atunci  $x\&\&y$  are valoarea 1, deoarece ambii operanzi sînt diferiți de zero. În schimb,  $x\&y$  are valoarea zero. Într-adevăr, în acest caz se realizează un *și pe biți* cu valorile:

```
x=0000000000000010
y=0000000000000001
-----
x&y=0000000000000000
```

Operatorul *sau exclusiv pe biți* se execută bit cu bit, conform tablei de mai jos:

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0
```

Operatorul  $\wedge$  are prioritatea imediat mai mică decît operatorul  $\&$ . Se utilizează pentru a anula sau poziționa diferiți biți.

**Exemple:**

1. `int a;`

`a ^ a`

are valoarea 0. Într-adevăr, cei doi operanzi fiind identici, se execută operatorul  $\wedge$  pentru biți identici, ori în acest caz el are ca rezultat pe zero.

2. `int a;`

`a ^ 1`

are o valoare care depinde de paritatea lui  $a$ . Într-adevăr, dacă  $a$  este par, atunci ultimul său bit este zero.

Expresia

`a ^ 1`

setează la valoarea unu ultimul bit a lui  $a$ . În felul acesta, rezultatul este egal cu cel mai mic număr impar care-l depășește pe  $a$ .

Dacă  $a$  este impar, atunci ultimul bit are valoarea unu.

Expresia

$$a \wedge 1$$

anulează ultimul bit a lui  $a$ . Deci, rezultatul este egal cu cel mai mare număr par care nu-l depășește pe  $a$ .

Operatorul *sau logic pe biți* se execută bit cu bit, conform tabelului de mai jos:

$$1 \mid 1 = 1$$

$$1 \mid 0 = 1$$

$$0 \mid 1 = 1$$

$$0 \mid 0 = 0$$

Acest operator are prioritatea imediat mai mică decât operatorul *sau exclusiv pe biți* ( $\wedge$ ) și imediat mai mare decât operatorul *și logic* ( $\&\&$ ).

Operatorul  $|$  se folosește la setări de biți.

**Exemple:**

1. `int a;`

$$a \mid 1$$

are o valoare a cărui ultim bit este unu, indiferent de valoarea variabilei  $a$ . Celalți biți coincid cu cei ai lui  $a$ . De aceea, valoarea acestei expresii este cel mai mic număr impar care nu este mai mic decât  $a$ .

2. `int a;`

$$a \mid 0100000$$

are o valoare cu bitul cel mai semnificativ setat, indiferent de valoarea variabilei  $a$ . Celalți biți coincid cu cei ai lui  $a$ .

### 3.2.7. Operatorii de atribuire

Operatorul de atribuire, în forma cea mai simplă, se notează prin caracterul `=`. El se utilizează în expresii de forma:

$$v = (\text{expresie})$$

unde:

- $v$  - Este o variabilă simplă, referențiază un element de tablou (variabilă cu indici) sau de structură.



Operatorul de atribuire are o prioritate mai mică decât toți operatorii pe care i-am întâlnit pînă în prezent. De aceea, parantezele din construcția indicată mai sus, de obicei nu sînt necesare.

Operatorul de atribuire are ca efect atribuirea valorii expresiei aflată în dreapta semnelui de atribuire variabilei  $v$ . Ulterior o să vedem că în stînga semnelui de atribuire se poate afla chiar o expresie care definește o adresă.

La o atribuire se face și o conversie dacă este necesar. Astfel, valoarea expresiei din dreapta semnelui de atribuire se va converti spre tipul variabilei din stînga lui înainte de a se face atribuirea, dacă cele două tipuri sînt diferite.

**Expresia**

$v = \text{expresie}$

este o *expresie de atribuire*. Ea are ca *valoare* valoarea care se atribuie, iar ca *tip*, tipul variabilei  $v$ .

Rezultă că o construcție de forma:

$v1 = (v = \text{expresie})$

este corectă și reprezintă tot o expresie de atribuire: lui  $v1$  i se atribuie valoarea atribuită în prealabil lui  $v$ , făcîndu-se și conversie dacă este necesar.

Deoarece operatorii de atribuire se asociază de la *dreapta spre stînga*, expresia de mai sus se poate scrie fără paranteze:

$v1 = v = \text{expresie}$

În general, o expresie de atribuire are forma

$v_n = \dots = v_1 = v = \text{expresie}$

Valoarea expresiei aflate în partea dreaptă se atribuie întîi lui  $v$ , apoi lui  $v1$  și așa mai departe și în final se atribuie lui  $v_n$ . Atribuiriile sînt precedate de conversii în cazul în care valoarea care se atribuie este de un tip diferit decât tipul variabilei la care se face atribuirea.

Pentru operatorii de atribuire, în afara semnelui  $=$  se mai poate folosi și succesiunea de caractere:

$op =$

unde  $op$  este un operator binar aritmetic sau logic pe biți. Deci  $op$  poate fi unul din operatorii:

$/ \% * - + < < > > \& \wedge |$

Această construcție se folosește pentru a face prescurtări.

Expresia

$v\ op = \text{expresie}$

este echivalentă cu expresia de atribuire:

$v = v\ op(\text{expresie})$

Expresiile de atribuire pot fi folosite peste tot în program unde este legal să apară o expresie. Acest fapt permite adesea să se facă compactări în programul sursă.

**Exemple:**

1.

`int a;`

`a=10`

variabilei *a* i se atribuie valoarea 10.

2.

`int i;`

`i=i+3`

valoarea variabilei *i* se mărește cu 3.

3.

`int i;`

`i += 3`

are același efect ca și expresia de la exemplul precedent.

4. Expresia de atribuire:

$a[i*3+10][j*2-3] = a[i*3+10][j*2-3]*x$

se scrie prescurtat astfel:

$a[i*3+10][j*2-3]*=x$

### Exerciții:

3.14 Să se scrie un program care citește valoarea lui  $x$ , calculează valorile expresiilor:

$$4x^2x + 3x$$

$$4x^2x + 3x + 1$$

și

$$(4x^2x + 3x - 1)/(4x^2x + 3x + 1)$$

și afișează valorile respective.

### PROGRAMUL BIII14

```
#include <stdio.h>
main() /* citește pe x și afișează valorile expresiilor:
    4x²x + 3x
    4x²x + 3x + 1
    (4x²x + 3x - 1)/(4x²x + 3x + 1) */
{
    double x,y;

    scanf("%lf", &x);
    printf("x=%g\ty=4x²x + 3x=%g\n",x,y=4*x*x + 3*x );
    printf("4x²x+3x+1 = %g\ty= (4x²x+3x-1)/(4x²x+3x+1)\n"
           = %g\n",y+1, (y-1)/(y+1));
}
```

### Observație:

Parametrul:

$$y=4x^2x+3x$$

din primul apel al funcției *printf* este o expresie de atribuire. Prin intermediul ei se atribuie lui  $y$  valoarea expresiei

$$4x^2x+3x$$

Valoarea expresiei de atribuire coincide cu valoarea atribuită lui  $y$ . Tipul expresiei de atribuire coincide cu tipul lui  $y$ , deci tipul ei este *double*.

3.15 Să se scrie un program care citește valoarea lui  $x$ , calculează și afișează valorile expresiilor:

$$x^{**10}, x^{**20}, x^{**30} \text{ și } x^{**40}.$$

## PROGRAMUL BIII15

```
#include <stdio.h>
#include <math.h>

main() /* citește pe x și afișează valorile expresiilor:
        x**10
        x**20
        x**30
        x**40 */
{
    double x,y,z;

    scanf("%lf", &x);
    printf("x=%g\tx**10 =%g\n", x, y = pow(x, 10.0));
    printf("\tx**20 =%g\n", z = y*y);
    printf("\tx**30 =%g\n", y*z);
    printf("\tx**40 =%g\n", z*z);
}
```

- 3.16 Să se scrie un program care citește valoarea lui  $x$ , afișează valorile lui  $x$ ,  $[x]$ ,  $\{x\}$ , calculează și afișează valorile expresiilor:

$$7[x] \cdot [x] - 3[x] + 10$$

și

$$7\{x\} \cdot \{x\} - 3\{x\} + 10$$

unde:

- $/x/$  - Notează partea întreagă a lui  $x$  ( $[x] \leq x$ ).  
 $\{x\}$  - Notează partea fracționară a lui  $x$  ( $\{x\} = x - [x]$ ).

## PROGRAMUL BIII16

```
#include <stdio.h>
main() /* - citește pe x;
        - afișează pe x, [x], {x};
        - calculează și afișează valorile expresiilor:
            7[x]*[x] - 3[x] + 10
            și
            7{x}*{x} - 3{x} + 10 */
{
    int i;
    double x,y;
```

```

scanf("%lf", &x);
printf("x=%g\t", x);
printf("[x]=%d\t", i = x);
printf("{x}=%f\n", y = x - 1);
printf("7[x][x] - 3[x] + 10=%d\n",
      7*i*i - 3*i + 10);
printf("7{x}{x} - 3{x} + 10=%g\n",
      7*y*y - 3*y + 10);
}

```

### 3.2.8. Operatorii de incrementare și decrementare

Acești operatori sînt unari și au aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

Operatorul de *incrementare* se notează cu ++, iar cel de *decrementare* cu --.

Operatorul de incrementare mărește valoarea operandului său cu 1, iar cel de decrementare micșorează valoarea operandului său cu 1.

Acești operatori pot fi folosiți *prefixați*.

++*operand*  
--*operand*

sau *postfixați*:

*operand*++  
*operand*--

În cazul în care un operator de incrementare sau decrementare este prefixat, se folosește valoarea operandului la care s-a aplicat operatorul respectiv.

În cazul în care un operator de incrementare sau decrementare este postfixat, se folosește valoarea operandului dinaintea aplicării operatorului respectiv.

**Exemplu:**

Presupunem că *x* are valoarea 3. Dacă considerăm expresia de atribuire  
*y* = ++*x*

atunci lui *y* i se atribuie valoarea 4 (la atribuire se folosește valoarea incrementată).

Dacă utilizăm expresia de atribuire

$y = x++$

pentru  $x=3$ , atunci lui  $y$  i se atribuie valoarea 3 (la atribuire se folosește valoarea dinaintea incrementării)

În ambele cazuri valoarea lui  $x$  s-a mărit cu 1.

### 3.2.9. Operatorul de forțare a tipului sau de conversie explicită (expresie cast)

Adesea dorim să specificăm conversia valorii unui operand spre un tip dat. Acest lucru este posibil folosind o construcție de forma:

*(tip) operand*

Printr-o astfel de construcție valoarea operandului se convertește spre tipul indicat în paranteze.

În construcția de mai sus (tip) se consideră că este un operator unar. Îl vom numi *operator de forțare a tipului sau de conversie explicită*.

Construcția de mai sus o vom numi *expresie cast*.

#### Exemplu:

Presupunem că funcția  $f$  are un parametru de tip *double*. Fie declarația:

```
int n;
```

Atunci, pentru a apela  $f$  cu parametrul  $n$ , este necesar ca, în prealabil,  $n$  să se convertească spre *double*. Acest lucru se poate realiza printr-o atribuire:

```
double x;
```

```
...
```

```
f(x=n);
```

Un alt mod mai simplu de conversie a parametrului întreg spre tipul *double* este utilizarea unei *expresii cast*:

```
f((double)n);
```

Operatorul *(tip)* fiind unar, are aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

#### Exerciții:

- 3.17 Să se scrie un program care citește un întreg și afișează rădăcina pătrată din numărul respectiv.

## PROGRAMUL BIII17

```
#include <stdio.h>
#include <math.h>

main() /* - citește pe n;
        - calculează și afișează rădăcina pătrată din n */
{
    long n;

    scanf("%ld", &n);
    printf("n=%ld\tsqrt(n)=%g\n", n, sqrt((double)n));
}
```

### Observație:

În acest program s-a utilizat funcția *sqrt* pentru extragerea rădăcinii pătrate. Ea are prototipul:

```
double sqrt(double);
```

Acest prototip este definit în fișierul *math.h*.

218 Să se scrie un program care citește pe  $n$  de tip întreg și afișează valoarea expresiei  $n/(n+1)$  cu 15 zecimale

## PROGRAMUL BIII18

```
#include <stdio.h>
main() /* - citește pe n;
        - calculează și afișează pe  $n/(n+1)$  cu 15 zecimale */
{
    long n;

    scanf("%ld", &n);
    printf("n=%ld\t $n/(n+1)=%.15g$ \n", n,
           (double)n/(n+1));
}
```

### Observație:

Expresia  $n/(n+1)$  reprezintă împărțirea întregă a lui  $n$  la  $n+1$ . Pentru a obține rezultatul pe 15 zecimale este necesar să se efectueze în părțile neîntregi. În acest scop s-a convertit operandul spre tipul *double*. În fapt,

acesta, conform regulei conversiilor implicite, se convertește spre *double* și cel de al doilea operand și apoi se face împărțirea celor doi operanzi flotanți.

### 3.2.10. Operatorul dimensiune

Dimensiunea în octeți a unei date sau al unui tip se poate determina folosind operatorul *sizeof*. El poate fi folosit sub forma

*sizeof data*

sau

*sizeof(tip)*

unde,

*data*

- Poate fi:

- un nume de variabilă simplă;
- un nume de tablou;
- un nume de structură,
- referirea la un element de tablou (variabilă cu indici);
- referirea la elementul unei structuri.

*tip*

- Poate fi:

- un cuvânt sau cuvintele cheie ale unui tip predefinit;
- o construcție care definește un tip.

**Expresia**

*sizeof data*

poate fi scrisă și sub forma *sizeof(data)*. Ea are ca valoare dimensiunea în octeți a operandului "data". Astfel, dacă data este:

*numele unei  
variabile simple*

- Atunci rezultatul va fi numărul de octeți alocați variabilei respective.

*numele unui  
tablou*

- Atunci rezultatul va fi numărul de octeți al zonei de memorie alocate tabloului respectiv.

*numele unei  
structuri*

- Atunci rezultatul va fi numărul de octeți al zonei de memorie alocate structurii respective.

*etc.*

**Expresia**



`sizeof(tip)`

are ca valoare numărul de octeți alocați pentru reprezentarea unei date de tip *tip*.

Operatorul dimensiune (*sizeof*) este unar și are aceeași prioritate ca restul operatorilor unari ai limbajului C.

**Exemple:**

1.

`int x;`

`sizeof x`

are valoarea 2, deoarece pentru variabila *x* se alocă 2 octeți.

2.

`long double y[7];`

`sizeof y[3]`

are valoarea 10, deoarece pentru o dată de tip *long double* se alocă 10 octeți.

`sizeof y`

are valoarea 70, deoarece tabloul are 7 elemente în total.

3.

`sizeof(char)`

are valoarea 1, deoarece pentru o dată de tip *char* se alocă un octet.

### 3.2.11. Operatorul adresă

Operatorul adresă este unar și se notează prin caracterul **&**. El se aplică pentru a determina adresa de început a zonei de memorie alocată unei date. În forma cea mai simplă, acest operator se utilizează în construcții de forma:

***&nume***

unde:

***nume*** - Este numele unei variabile simple sau al unei structuri.

Menționăm că în cazul în care *nume* este numele unui tablou, atunci acesta are ca valoare chiar adresa de început a zonei de memorie alocată

tabloului respectiv. Deci în acest caz nu se mai utilizează operatorul adresa &.

Acest operator a fost utilizat deja frecvent la apelul funcției scanf.

Mai târziu o să vedem și alte utilizări ale acestui operator.

Fiind un operator unar, el are acces și la variabile ca și ceilalți operatori unari ai limbajului C.

Operatorul unar & are în limbajul C ++ și o altă utilizare așa cum o va vedea mai târziu.

### 3.2.12. Operatorii paranteză

Parantezele rotunde se utilizează în principal pentru a gruparea expresiei, fie la apelul funcțiilor.

Amintim că o expresie în limbajul paranteze rotunde folosește un operand. În felul acesta se poate opune o altă ordine în efectuarea operațiilor, decât cea care rezultă din prioritatea și asociativitatea operatorilor.

Operații obișnuite pentru funcțiile unei expresii în paranteze înțeleg anumite limite asupra operatorilor. De exemplu, la un astfel de operand nu se pot afla operandii de instrumentare și declarare, de exemplu adresa. Astfel construcțiile.

$(i+10)++$   $--(x+y)$   $\&(x+y)$

sunt eronate.

La apelul unei funcții la parametrii ei care se adresează în paranteze rotunde. În acest caz se obișnuiește să se spună că parantezele rotunde sunt *operatori de apel de funcție*.

Parantezele patrate includ expresii care reprezintă *index*. Ele sunt numiți *operatori de indexare*.

Parantezele sunt operatorii primare în limbajul C, iar la un ar, au prioritatea imediat mai mică decât paranteze.

### 3.2.13. Operatorii condiționali

Operatorii condiționali permit construirea de expresii care să aibă o valoare depindă de valoarea unei condiții.

Prin *condiție* înțelegem o expresie care poate avea valoarea *adevărată* sau *falsă*. În limbajul C, o condiție este reprezentată printr-o expresie care

El are valoarea adevărată dacă este derivată de zero și fals în caz contrar.

Numim *expresie condițională*, o expresie a cărei valoare și tip este dependentă de valoarea unei condiții.

Un exemplu simplu de expresie condițională este aceea care are ca valoare maximul dintre două numere. Astfel, dacă  $a$  și  $b$  sînt două numere, atunci valoarea expresiei care calculează maximul dintre  $a$  și  $b$  depinde de valoarea condiției  $a > b$ .

Într-adevăr, dacă  $a > b$  are valoarea adevărată, atunci expresia respectivă are valoarea  $a$ . În caz contrar (adică  $a > b$  are valoarea falsă) expresia de calcul a maximului va avea valoarea  $b$ .

O expresie condițională are formatul:

$E1 ? E2 : E3$

unde:

$E1, E2$  și  $E3$  - Sînt expresii

Această expresie se evaluează astfel:

- Se determină valoarea expresiei  $E1$ .
- Dacă  $E1$  are o valoare diferită de zero (are valoarea adevărată), atunci valoarea și tipul expresiei condiționale coincide cu valoarea și tipul expresiei  $E2$ . Altfel (adică  $E1$  are valoarea zero) valoarea și tipul expresiei condiționale coincide cu valoarea și tipul expresiei  $E3$ .

De aici rezultă că valoarea și tipul expresiei condiționale depinde de valoarea expresiei  $E1$ .

Folosind formatul de mai sus, calculul maximului dintre  $a$  și  $b$  se exprimă cu ajutorul expresiei condiționale:

$(a > b) ? a : b$

unde:

- $(a > b)$  - Este expresia  $E1$  (condiția).
- $a$  - Este expresia  $E2$ .
- $b$  - Este expresia  $E3$ .

Operatorii condiționali sînt:

$?$  și  $:$

Ei se folosesc totdeauna împreună și în ordinea indicată în formatul expresiei condiționale.

Ei au prioritatea imediat mai mică decât prioritatea operatorului *sau* *logic* (`||`) și imediat mai mare decât prioritatea operatorilor de atribuire. Ținând seama de prioritatea operatorilor, rezultă că se pot omite parantezele din expresia pentru calculul maximului dintre *a* și *b*:

`a > b ? a : b`

Expresia condițională este un caz particular de expresie și deci ea poate fi utilizată oriunde este legal să apară, în program, o expresie.

De exemplu:

`v = (E1 ? E2 : E3)`

este o expresie de atribuire: în partea dreaptă a semnului de atribuire se află o expresie condițională. Ea poate fi scrisă mai simplu fără paranteze, deoarece operatorul de atribuire este mai puțin prioritar decât operatorii condiționali.

Operatorii condiționali se asociază de la dreapta la stînga la fel ca și operatorii unari și de atribuire.

**Exerciții:**

3.19 Să se scrie un program care citește două numere și afișează maximumul dintre ele.

### PROGRAMUL BIII19

```
#include <stdio.h>
main() /* citește doua numere si afiseaza maximumul dintre ele */
{
    double a,b;

    scanf("%lf %lf", &a,&b);
    printf("a=%g\tb=%g\tmax(a,b)=%g\n", a,b,
        a > b ? a : b );
}
```

3.20 Să se scrie un program care citește un număr și afișează valoarea lui absolută.

### PROGRAMUL BIII20

```
#include <stdio.h>
main() /* citește un numar si afiseaza valoarea lui absoluta */
{
```

```
double a;

scanf("%lf", &a);
printf("a=%g\tabs(a)=%g\n", a, a < 0 ? -a : a );
}
```

- 3.21 Să se scrie un program care citește doi întregi și afișează maximul dintre valorile lor absolute.

### PROGRAMUL BIII21

```
#include <stdio.h>
main() /* citește doua numere întregi și afișează maximul dintre
        valorile lor absolute */
{
    int a,b,c,d;

    scanf("%d %d", &a,&b);
    printf("a=%d\tb=%d\tabs(a)=%d\tabs(b)=%d\n",a,b,
           c =a<0 ? -a : a, d=b < 0 ? -b : b);
    printf("max(abs(a),abs(b))=%d\n",c > d ? c : d );
}
```

- 3.22 Să se scrie un program care citește valoarea variabilei  $x$  și afișează valoarea funcției  $f(x)$  definită ca mai jos:

$$f(x) = \begin{cases} 3x^2 + 7x - 10 & \text{pentru } x < 0; \\ 2 & \text{pentru } x = 0; \\ 4x^2 - 8 & \text{pentru } x > 0. \end{cases}$$

### PROGRAMUL BIII22

```
#include <stdio.h>
main() /* citește pe x și afișează valoarea funcției f(x) definită astfel:
        3x^2 + 7x - 10  pentru x < 0;
        2              pentru x = 0;
        4x^2 - 8       pentru x > 0 */
{
    double x;

    scanf("%lf", &x);
    printf("x=%g\tf(x)=%g\n",x,
           x<0 ? 3*x*x+7*x-10 : x>0 ? 4*x*x-8:2);
}
```

### 3.2.14. Operatorul virgulă

Operatorul virgulă leagă două expresii în una singură conform formatului de mai jos:

*exp1,exp2*

Operatorul virgulă are cea mai mică prioritate dintre toți operatorii limbajului C. Prioritatea lui este imediat mai mică decât a operatorilor de atribuire.

Construcția de mai sus este o expresie. Valoarea și tipul acestei expresii coincide cu valoarea și tipul ultimei expresii, deci în cazul de față cu al lui *exp2*.

O construcție de forma:

*(exp1,exp2),exp3*

este corectă și ea este o nouă expresie. Într-adevar, *exp1, exp2* fiind expresii, *(exp1,exp2)* este un operand, deci tot o expresie, așa că după ea poate fi scrisă o virgulă urmată de o altă expresie. Deoarece operatorul virgulă are prioritatea cea mai mică, rezultă că parantezele pot fi omise, deci expresia de mai sus se poate scrie astfel:

*exp1,exp2,exp3*

În general, o construcție de forma:

*exp1,exp2,...,expn*

unde:

*exp1,exp2,...,expn* - Sînt expresii.

este o expresia a cărei valoare și tip coincide cu valoarea și tipul lui *expn*.

Într-o astfel de construcție, expresiile se evaluează pe rînd, de la stînga la dreapta.

Operatorul virgulă se utilizează în situații în care într-un anumit punct al unui program în care este legal să folosim o expresie, este necesar să se realizeze un calcul complex exprimat prin mai multe expresii.

#### Exerciții:

3.23 Să se scrie un program care citește doi întregi și afișează maximum dintre valorile lor absolute.

Acest exercițiu a fost rezolvat și în paragraful precedent (exercițiul 3.21.)

## PROGRAMUL BIII23

```
#include <stdio.h>
main() /* citește doi întregi și afișează maximumul dintre valorile
        lor absolute */
{
    int a,b,c,d;

    scanf("%d%d", &a,&b);
    printf("a=%d\tb=%d\tmax( abs(a),abs(b) )=%d\n",
        a,b, ((c=a<0?-a:a),
            (d=b<0 ? -b : b), (c > d)) ? c : d );
}
```

**Observație:**

**Expresia:**

$(c=a<0?-a),(d=b<0?-b),(c>d)$

se compune din trei expresii care se evaluează de la stînga la dreapta.

Prima atribuire lui  $c$  valoarea absolută a lui  $a$ , a doua atribuire lui  $d$  valoarea absolută a lui  $b$ , iar a treia testează relația  $c>d$ . Valoarea întregii expresii coincide cu 1 dacă  $c>d$  și cu zero în caz contrar.

### 3.2.15. Alți operatori ai limbajului C

În limbajul C se utilizează și alți operatori și anume:

- operatorul \* unar;

și

- operatorii . și ->.

Operatorul \* unar se utilizează pentru a face acces la conținutul unei zone de memorie delimitată prin adresa ei de început. Se obișnuiește să se spună că operatorul adresă (& unar) este *operator de referențiere*, iar operatorul \* unar este *operator de dereferențiere*. Acest operator va fi studiat ulterior împreună cu datele de tip adresă (*pointer*). El are aceeași prioritate ca și ceilalți operatori unari ai limbajului C.

Operatorii . și -> (acest simbol este compus din semnul "minus" urmat de semnul "mai mare") se utilizează pentru a se face acces la componentele unei structuri. Ei sînt de prioritate maximă, avînd aceeași prioritate cu parantezele. Ei vor fi studiați în capitolul cu privire la structuri.

### 3.2.16. Tabela cu prioritățile operatorilor limbajului C

În tabela de mai jos se indică operatorii limbajului C în ordinea descrescătoare a priorităților lor.

Operatorii din aceeași linie au aceeași prioritate. Ei se asociază de la stînga spre dreapta, exceptînd operatorii unari, condiționali și de atribuire, care se asociază de la dreapta spre stînga.

()	[]	,	->	++	--	(tip)	sizeof	!	~
+(unar)	-(unar)	&(unar)	*(unar)						
*(binar)	/	%							
+(binar)	-(binar)								
<<	>>								
<	<=	>=	>						
=	!=								
&(binar)									
^									
&&									
?	:								
=	<<=	>>=	+=	-=	*=	/=	%=	&=	^=
									=

Ulterior această tabelă va fi completată și cu alți operatori specifici limbajului C++.



## 4. INSTRUCȚIUNI

Am văzut că o funcție are structura:

*antei*  
*corp*

Corpul unei funcții conține, între acolade, o succesiune de declarații urmate de o succesiune de instrucțiuni:

```
{  
  declarații  
  instrucțiuni  
}
```

Corpul poate conține numai instrucțiuni sau numai declarații, sau se poate reduce la cele două acolade (acoladele vor fi totdeauna prezente).

*Declarațiile* permit utilizatorului să definească date de diferite tipuri (predefinite sau definite de utilizator). De asemenea, datele pot fi inițializate cu ajutorul declarațiilor.

Prelucrarea datelor se realizează cu ajutorul *instrucțiunilor*.

Ordinea în care se execută instrucțiunile unui program definește așa numita *structură de control a programului*.

Cea mai simplă structură de control este structura *secvențială*. O astfel de structură de control se compune dintr-o succesiune de instrucțiuni care se execută una după alta, în ordinea în care sînt scrise în program.

De obicei, la descrierea unui proces de calcul este nevoie să se utilizeze și alte tipuri de structuri. Astfel, C. Bohm și G. Jacopini au arătat în lucrarea [1], că pentru exprimarea proceselor de calcul sînt suficiente trei structuri de control și anume:

- structura secvențială;
- structura alternativă;

și

- structura repetitivă (ciclică) condiționată anterior.

Acest rezultat s-a aflat la baza ideii care a condus în anii '70 la conceptul de *programare structurată*. Acest concept a fost dezvoltat de E.W. Dijkstra în lucrările sale, iar ulterior și de alți specialiști, ca de exemplu, N. Wirth și C.A.R. Hoare. El reprezintă un *stil* în programare care se impune și în prezent. Un efect imediat al programării structurate este ridicarea produc-

ivităţii în programare şi creşterea fiabilităţii programelor.

Prin *program structurat* înţelegem un program care are o structură de control realizată numai cu ajutorul celor trei structuri amintite mai sus.

Ulterior s-au admis încă două structuri şi anume

- structura selectivă;

şi

structura repetitivă (ciclică) condiţionată posterior.

Introducerea acestor structuri permite o flexibilitate mai mare în programare. În acest fel, programarea structurată reprezintă un stil în programare care contribuie la realizarea de programe care au o structură clară şi care pot fi uşor de panate şi întreţinute.

Limbaajul C a fost prevăzut cu instrucţiuni menite să permită realizarea simplă a structurilor propriu programării structurate. Structura secvenţială se realizează cu ajutorul instrucţiunii *compose*, structura alternativă cu ajutorul instrucţiunii *if*, structura repetitivă condiţionată anterior, prin intermediul instrucţiunilor *while* şi *for*, structura selectivă se realizează cu ajutorul instrucţiunii *switch*, iar structura repetitivă condiţionată posterior cu ajutorul instrucţiunii *do-while*.

Limbaajul C are şi alte instrucţiuni care reprezintă elemente de bază în construirea structurilor amintite mai sus. Astfel de instrucţiuni sînt: instrucţiunea *expresie* şi instrucţiunea *vidă*.

Alte instrucţiuni prezente în limbaaj asigură o flexibilitate mare în programare. Acestea sînt instrucţiunile:

*return*, *break*, *continue* şi *goto*.

Mai jos se descriu aceste instrucţiuni şi se dau exemple simple de utilizare.

## 4.1. Instrucţiunea vidă

Instrucţiunea vidă se reduce la caracterul punct şi virgulă (;). Ea nu are nici un efect.

Instrucţiunea vidă se utilizează în construcţii în care se cere prezenţa unei instrucţiuni, dar nu trebuie să se execute nimic în punctul respectiv. Astfel de situaţii apar frecvent în cadrul structurilor alternative şi repetitive, aşa cum se va vedea în continuare.

## 4.2. Instrucțiunea expresie

*Instrucțiunea expresie* se obține scriind punct și virgulă după o expresie. Deci, instrucțiunea expresie are formatul:

expresie;

În cazul în care expresia din compunerea unei instrucțiuni expresie este o expresie de atribuire, se spune că instrucțiunea respectivă este o *instrucțiune de atribuire*.

Un alt caz utilizat frecvent este acela când expresia este un operand ce reprezintă apelul unei funcții. În acest caz, instrucțiunea expresie este o *instrucțiune de apel a funcției* respective.

**Exemple:**

1.

```
int x;  
...  
x=10;
```

Este o instrucțiune de atribuire. Variabilei *x* i se atribuie valoarea 10

2.

```
double y;  
..  
y=y+4; sau y+=4;
```

Este o instrucțiune de atribuire, care mărește cu 4 valoarea lui *y*

3.

```
putch(c-'a'+'A');
```

Este o instrucțiune de apel a funcției *putch*.

4.

```
double a;  
...  
a++;
```

Este o instrucțiune expresie, care mărește valoarea lui *a* cu unu.

5.

```
double a;  
...  
++a;
```

Are același efect ca și instrucțiunea expresie din exemplul precedent.

#### Observație:

Nu orice expresie urmată de punct și virgulă formează o instrucțiune expresie efectivă. De exemplu, construcția:

```
a;
```

deși este o instrucțiune expresie, ea nu are nici un efect.

#### Exerciții:

- 4.1 Să se scrie un program care citește trei întregi și afișează maximum dintre ei.

#### PROGRAMUL BIV1

```
#include <stdio.h>  
main() /* citește trei întregi și afișează maximum dintre ei */  
{  
    int a,b,c,d;  
  
    scanf("%d %d %d", &a,&b,&c);  
    d = a > b ? a : b; // d=max(a,b)  
    printf("a=%d\tb=%d\tc=%d\tmax(a,b,c)=%d\n",a,b,  
           c,d>c ? d : c);  
}
```

- 4.2 Să se scrie un program care citește valorile variabilelor  $a, b, c, d$ ,  $x$  de tip *double* și afișează valoarea expresiei:

$$(a^2x^2 + b^2x + c)/(a^2x^2 + b^2x + d)$$

dacă numitorul este diferit de zero și zero în caz contrar.

#### PROGRAMUL BIV2

```
#include <stdio.h>  
main() /* citește pe a,b,c,d și x, calculează și afișează  
        valoarea expresiei:  
        (a^2x^2 + b^2x + c)/(a^2x^2 + b^2x + d)
```

```

        daca numitorul este diferit de zero si zero în caz contrar */
{
    double a,b,c,d,x;
    double y,z,u;

    printf("a=");
    scanf("%lf",&a);
    printf("b=");
    scanf("%lf",&b);
    printf("c=");
    scanf("%lf",&c);
    printf("d=");
    scanf("%lf",&d);
    printf("x=");
    scanf("%lf",&x);
    y = a*x*x;
    z=b*x;
    u=y*x+z+d;
    printf("(a*x*x+b*x+c)/(a*x*x*x+b*x+d)=%g\n",
           u ? (y+z+c)/u : u);
}

```

### 4.3. Instrucțiunea compusă

*Instrucțiunea compusă* este o succesiune de instrucțiuni incluse între acolade, succesiune care poate fi precedată și de declarații:

```

{
    declarații
    instrucțiuni
}

```

Dacă declarațiile sînt prezente, atunci ele definesc variabile care sînt definite atît timp cit controlul programului se află la o instrucțiune din componerea instrucțiunii compuse.

#### Exemplu:

Presupunem că într-un anumit punct al programului este necesar să se permută valorile variabilelor întregi a și b. Aceasta se poate realiza astfel:

```

{
    int t;

    t=a;

```

```

a=b;
b=t;
}

```

Variabila *t* este definită de îndată ce controlul programului ajunge la prima instrucțiune din instrucțiunea compusă (*t=a*.) După execuția ultimei instrucțiuni a instrucțiunii compuse, variabila *t* nu mai este definită.

Instrucțiunea compusă se utilizează unde este necesară prezența unei instrucțiuni dar procesul de calcul din punctul respectiv este mai complex și se exprimă prin mai multe instrucțiuni. În acest caz instrucțiunile respective se includ între acolade pentru a forma o instrucțiune compusă.

Acest procedeu de a forma o instrucțiune compusă din mai multe instrucțiuni, se utilizează frecvent în construirea structurilor alternative și ciclice.

## 4.4. Instrucțiunea if

Instrucțiunea *if* are următoarele formate.

format 1

```

If(expresie)
    instrucțiune

```

format 2

```

If(expresie)
    instrucțiune1
else
    instrucțiune2

```

La întâlnirea instrucțiunii *if* întâi se evaluează expresia din paranteze. Apoi, în cazul formatului 1, dacă expresia are valoarea diferită de zero (adică are valoarea adevărat), atunci se execută *instrucțiune*; altfel se trece în secvență la instrucțiunea următoare instrucțiunii *if*. În cazul formatului 2, dacă expresia are o valoare diferită de zero, atunci se execută *instrucțiune 1* și apoi se trece în secvență (adică la instrucțiunea aflată după *instrucțiune 2*); altfel se execută *instrucțiune 2*.

În mod normal, în ambele formate după execuția instrucțiunii *if* se ajunge la instrucțiunea următoare ei. Cu toate acestea, este posibilă și o altă situație când instrucțiunile din compunerea lui *if*, definesc ele însele un alt mod de continuare a execuției programului.

Deoarece o instrucțiune compusă este considerată ca fiind un caz particular de instrucțiune, rezultă că instrucțiunile din compunerea lui *if* pot fi instrucțiuni compuse. De asemenea, instrucțiunile respective pot fi chiar instrucțiuni *if*. În acest caz se spune că instrucțiunile *if* sînt *imbriate*.

#### Exerciții:

#### 4.3 Se dă funcția:

$$y = 3 \cdot x^2 + 2 \cdot x - 10 \text{ pentru } x > 0$$

și

$$y = 5 \cdot x + 10 \text{ pentru } x \leq 0$$

Se se scrie un program care citește valoarea lui *x*, calculează și afișează valoarea lui *y*. Acest proces de calcul implică două alternative, în funcție de valoarea lui *x*:

dacă  $x > 0$ , atunci  $y = 3 \cdot x^2 + 2 \cdot x - 10$

altfel (adică  $x \leq 0$ ), atunci  $y = 5 \cdot x + 10$ .

Aceasta se transcrie imediat, în limbajul C, folosind formatul 2 al instrucțiunii *if*:

```
if (x > 0, /* x > 0 este expresie din formatul 2 */
    y = 3 * x * x + 2 * x - 10; /* instrucțiune 1 */
else
    y = 5 * x + 10; /* instrucțiune 2 */
```

Același proces de calcul se poate realiza cu ajutorul expresiei de atribuire

```
y = x > 0 ? 3 * x * x + 2 * x - 10 : 5 * x + 10;
```

care utilizează în partea dreaptă o expresie condițională. Aceasta este o scriere mai compactă, dar nu atât de evidentă ca și instrucțiunea *if* de mai sus.

#### PROGRAMUL BIV3

```
#include <stdio.h>
main() /* citește pe x, calculează și afișează valoarea lui
        y definită astfel:
            y = 3x^2 + 2x - 10  dacă x > 0
            y = 5x + 10       dacă x ≤ 0 */
{
    float x,y;
```

```
scanf("%f",&x);
if(x > 0)
    y= 3*x*x + 2*x -10;
else
    y= 5*x + 10;
printf("x= %f\ty= %f\n", x,y);
}
```

- 4.4 Să se scrie un program care citește valoarea lui  $x$ , calculează și afișează valoarea lui  $y$  definită ca mai jos:

$y=4x^3+5x^2-2x+1$     pentru  $x<0$ ;  
 $y=100$     pentru  $x=0$ ;  
 $y=2x^2+8x-1$     pentru  $x>0$ .

Acest proces de calcul implică următoarele alternative:

dacă  $x < 0$  atunci  $y=4x^3+5x^2-2x+1$

altfel

    dacă  $x=0$  atunci  $y=100$

    altfel

$y=2x^2+8x-1$

El se transcrie imediat printr-o instrucțiune *if imbricată*:

```
if (x<0)                /* x<0 este expresie din formatul 2 */
    y=4*x**3+5*x*x-2*x+1; //instrucțiune 1
else
    //instrucțiune 2 este instrucțiune if
    if (x==0)
        y=100;
    else
        y=2*x*x+8*x-1;
```

Același proces de calcul se poate descrie prin instrucțiunea expresie de mai jos:

$y=x<0?4x^3+5x^2-2x+1:x==0?100:2x^2+8x-1;$

## PROGRAMUL BIV4

```
#include <stdio.h>
main() /* citește pe x, calculează și afișează pe y definit astfel:
```



```

        y = 4*x*x*x + 5*x*x - 2*x + 1  daca x<0;
        y = 100                        -  daca x=0;
        y = 2*x*x + 8*x - 1            altfel */
{
    float x,y,a;

    scanf("%f", &x);
    a = x*x;
    if( x < 0 )
        y = 4*x*a + 5*a - 2*x + 1;
    else
        if( x == 0 )
            y = 100;
        else
            y = 2*a + 8*x - 1;
    printf("x=%f\ty=%f\n", x,y);
}

```

- 4.5 Să se scrie un program care citește valorile variabilelor neîntregi  $a$  și  $b$ , calculează rădăcina ecuației:

$$ax + b = 0$$

și afișează rezultatul.

Procesul de calcul al rădăcinii ecuației de mai sus trebuie să verifice existența ei:

dacă  $a$  este diferit de zero

atunci  $x = -b/a$

altfel

dacă  $b = 0$

atunci ecuația este *nedeterminată*;

altfel ecuația *nu are soluție*.

## PROGRAMUL BIV5

```

#include <stdio.h>
main() /* citește pe a și b, calculează și afișează rădăcina ecuației
        ax + b = 0 */
{
    double a,b;

```

```

if(scanf("%lf %lf", &a,&b) != 2)
    printf("coeficienti eronati\n");
else
    if ( a != 0)
        printf("a=%g\tb=%g\tx=%g\n",a,b, -b/a);
    else
        if( b== 0)
            printf("ecuatie nedeterminata\n");
        else
            printf("ecuatia nu are solutie\n");
}

```

### Observații:

1. Programul de față conține o instrucțiune *if* imbricată.
2. Pentru a mări claritatea programelor se obișnuiește să se decaleze spre dreapta (ca un tabulator) instrucțiunile din componența unei instrucțiuni *if*.
3. În acest program s-a realizat test relativ la valorile de date pentru *a* și *b*. Dacă funcția *scanf* nu returnează valoarea 2, înseamnă că nu s-au putut citi două numere la terminal. De aceea, într-o astfel de situație se afișează mesajul:

"coeficienti eronati".

- 4.6. Să se scrie un program care citește coeficienții *a, b, c, d, e, f*, ai unui sistem de două ecuații liniare cu două necunoscute, determină și afișează soluția acestuia când are o soluție unică.

Fie sistemul de ecuații liniare:

$$ax+by=c$$

$$dx+ey=f$$

Notăm cu *det* determinantul coeficienților necunoscutelor. Acesta se calculează cu relația:

$$det=ae-bd$$

Notăm cu *det1* determinantul obținut din *det* prin înlocuirea primei coloane cu coloana termenului liber și cu *det2* determinantul obținut din *ax* prin înlocuirea coloanei a doua cu coloana termenului liber. Atunci

$$det1=ce-bf \text{ și } det2=af-cd.$$

Procesul de calcul al valorilor lui *x* și *y* se desfășoară conform pașilor de

mai jos:

1. Se citesc valorile coeficienților

$a, b, c, d, e, f$ .

2.  $\det = ae - bd$

3. Dacă  $\det = 0$ , atunci sistemul nu are soluție unică (este nedeterminat sau incompatibil).

Se afișează în mesaj corespunzător și se întrerupe execuția programului.

Altfel se continuă cu punctul 4.

4.  $\det1 = ce - bf$ .

5.  $\det2 = af - cd$

6.  $x = \det1 / \det$ .

7.  $y = \det2 / \det$

8. Se afișează valorile lui  $x$  și  $y$ .

## PROGRAMUL BIV6

```
#include <stdio.h>
```

```
main() /* citește pe a,b,c,d,e,f, determina și afișează soluția sistemului
```

```
ax + by = c
```

```
dx + ey = f
```

```
în cazul în care are soluție unică */
```

```
{
```

```
double a,b,c,d,e,f,x,y,det,det1,det2;
```

```
if(scanf("%lf %lf %lf %lf %lf %lf",&a,
```

```
&b,&c,&d,&e,&f) !=6)
```

```
printf("coeficienți eronați\n");
```

```
else
```

```
if((det = a*e - b*d) == 0)
```

```
printf("sistemul are determinantul nul\n");
```

```
else
```

```
{
```

```
det1 = c*e - b*f;
```

```
det2 = a*f - c*d;
```

```
x = det1/det;
```

```
y = det2/det;
```

```

        printf("x=%g\ty=%g\n",x,y);
    }
}

```

#### Observații:

##### 1. Expresia

$$(1) (det=a*e-b*d) == 0$$

se evaluează astfel:

##### a. Se calculează

$$a*e-b*d$$

și valoarea respectivă se atribuie lui *det*.

##### b. Se compară valoarea atribuită lui *det* cu zero.

Dacă valoarea lui *det* este zero, atunci expresia (1) are valoarea *adevărat*; în caz contrar, expresia (1) are valoarea zero, adică *fals*.

##### 2. Parantezele din expresia (1) sînt obligatorii.

În lipsa lor, expresia (1) devine:

$$(2) det=a*e-b*d==0.$$

Aceasta se evaluează astfel:

##### a. Se calculează valoarea expresiei:

$$a*e-b*d;$$

##### b. Deoarece operatorul $==$ este mai prioritar decît $=$ , se compară valoarea expresiei respective cu zero.

Dacă expresia respectivă are valoarea zero, atunci expresia

$$(3) a*e-b*d==0$$

are valoarea 1 (*adevărat*),

altfel are valoarea zero (*fals*).

##### c. Se atribuie lui *det* valoarea expresiei 3, adică 0 sau 1.

#### 4.7 Să se scrie un program care citește valorile variabilelor *a*, *b*, *c*, calculează și afișează rădăcinile ecuației de gradul 2:

$$a*x^2+b*x+c=0$$

Pașii procesului de calcul sînt:

1. Se citesc valorile variabilelor  $a, b, c$ .
2. Dacă  $a=b=c=0$ , ecuația este nedeterminată.  
Se afișează un mesaj corespunzător și se întrerupe execuția programului.
3. Dacă  $a=b=0$  și  $c$  este diferit de zero, ecuația nu are soluție.  
Se afișează un mesaj corespunzător și se întrerupe execuția programului.
4. Dacă  $a=0$  și  $b$  este diferit de zero, ecuația se reduce la o ecuație de gradul întâi a cărei soluție este:  
$$x = -c/b;$$
se afișează un mesaj corespunzător, soluția  $x$  și se întrerupe execuția programului.
5. Dacă  $a$  este diferit de zero, se calculează:  
$$\text{delta} = b^2 - 4 \cdot a \cdot c$$
și  
$$d = 2 \cdot a.$$
6. Dacă  $\text{delta} > 0$ , atunci  $\text{delta} = \text{sqrt}(\text{delta})$ ; (*sqrt* este funcția de bibliotecă prin care se calculează rădăcina pătrată).  
Se determină și se afișează rădăcinile:  
$$x1 = (-b + \text{delta})/d;$$
$$x2 = (-b - \text{delta})/d;$$
apoi se întrerupe execuția programului,
7. Dacă  $\text{delta} = 0$ , atunci se determină și se afișează rădăcina dublă:  
$$x = -b/d;$$
se întrerupe execuția programului.
8. Altfel, ecuația are rădăcini complexe conjugate.  
Se calculează:  
$$\text{delta} = \text{sqrt}(-\text{delta}).$$
Se determină și se afișează rădăcinile:  
$$x1 = -b/d + i \cdot \text{delta}/d;$$
$$x2 = -b/d - i \cdot \text{delta}/d.$$

Se întrerupe execuția programului.

## PROGRAMUL BIV7

```
#include <stdio.h>
#include <math.h> //astfel conține prototip pentru sqrt

main() /* citește pe a,b,c, calculează și afișează rădăcinile ecuației:
        a*x*x + b*x + c = 0 */
{
    double a,b,c,d,delta;

    printf("coeficientul lui x pătrat:");
    if(scanf("%lf",&a) != 1)
        printf("coeficientul lui x pătrat eronat\n");
    else /* 1 */
        printf("coeficientul lui x:");
        if(scanf("%lf",&b) != 1)
            printf("coeficientul lui x eronat\n");
        else /* 2 */
            printf("termenul liber:");
            if(scanf("%lf",&c) != 1)
                printf("termenul liber eronat\n");
            else /* 3 */
                /* afișează coeficienții citiți */
                printf("a=%g\tb=%g\tc=%g\n",a,b,c);
                if( a == 0 && b == 0 && c == 0)
                    printf("ecuație nedeterminată\n");
                else
                    if( a == 0 && b == 0)
                        printf("ecuația nu are\
                            soluție\n");
                    else
                        if ( a == 0 ) {
                            printf("ecuație de gradul\
                                1\n");
                            printf("x=%g\n",-c/b);
                        }
                        else /* 4 */ /* a != 0 */
                            delta = b*b - 4*a*c;
                            d = 2*a;
                            if( delta > 0 ) {
                                printf("ecuația are 2\
```

```

        radacini reale si distincte\n');
        delta = sqrt(delta);
        printf("x1=%g\tx2=%g\n",
            (-b + delta)/d,
            (-b - delta)/d);
    }
    else
        if( delta == 0 ) {
            printf("ecuatia are\
                radacina dubla\n");
            printf("x=%g\n", -b/d);
        }
        else { /* 5 */
            printf("radacini\
                complexe\n");
            delta = sqrt(-delta)/d;
            d = -b/d;
            printf("x1=%g+i(%g)\n",d,
                delta);
            printf("x2=%g-i(%g)\n",d,
                delta);
        } /* sfirsit*5**/
    } /* sfirsit*4**/
} /* sfirsit*3**/
} /* sfirsit*2**/
} /* sfirsit*1**/
} /* sfirsit program*/

```

#### Observație:

Programul de față utilizează instrucțiunea *if* cu nivel de imbricare relativ mare. Programele care folosesc instrucțiuni *if* cu nivele mari de imbricare sînt greu de urmărit. Adesea se uită închiderea tuturor acoladelor. De aceea se recomandă reducerea pe cît posibil a nivelelor de imbricare ale instrucțiunilor *if*. În multe cazuri, nivelul de imbricare al unei instrucțiuni *if* se poate reduce folosind funcția *exit* (vezi paragraful următor)

## 4.5. Funcția standard *exit*

Funcția *exit* are prototipul:

```
void exit(int cod)
```

și el se află în fișierele de tip *h*:

**stdlib.h**

și

**process.h**

La apelul acestei funcții au loc următoarele acțiuni:

- se videază zonele tampon (bufferele) ale fișierelor deschise în scriere;
- se închid toate fișierele deschise;
- se intrerupe execuția programului.

Parametrul acestei funcții definește starea programului la momentul apelului.

Valoarea zero definește o terminare normală a execuției programului, iar o valoare diferită de zero semnalează prezența unei erori (terminarea anormală a execuției programului).

În concluzie, putem apela funcția *exit* pentru a termina execuția unui program, indiferent de faptul că acesta se termină normal sau din cauza unei erori.

**Exerciții:**

- 4.8 Să se modifice programul din exercițiul 4.7 în așa fel încât el să nu mai conțină instrucțiuni *if* imbricate.

## PROGRAMUL BIV8

```
#include <stdio.h>
#include <math.h> // fisierul contine prototip pentru sqrt
#include <stdlib.h> // fisierul contine prototip pentru exit

main() /* citește pe a,b,c, calculează și afișează rădăcinile ecuației:
      a*x*x + b*x + c = 0 */
{
    double a,b,c,d,delta;

    printf("coeficientul lui x patrat:");
    if(scanf("%lf",&a) != 1) {
        printf("coeficientul lui x patrat eronat\n");
        exit(1); /* terminare la eroare */
    }
    printf("coeficientul lui x:");
```



```

if(scanf("%lf",&b) != 1) {
    printf("coeficientul lui x eronat\n");
    exit(1);/* terminare la eroare */
}
printf("termenul liber:");
if(scanf("%lf", &c) != 1) {
    printf("termenul liber eronat\n");
    exit(1);/* terminare la eroare */
}

/* afiseaza coeficientii cititi */
printf("a=%g\tb=%g\tc=%g\n",a,b,c);
if( a== 0 && b == 0 && c == 0) {
    printf("ecuatie nedeterminata\n");
    exit(0);/* terminare fara erori */
}
if( a == 0 && b == 0){
    printf("ecuatia nu are solutie\n");
    exit(0);
}
if ( a == 0 ) {
    printf("ecuatie de gradul 1\n");
    printf("x=%g\n",-c/b);
    exit(0);
}
delta = b*b - 4*a*c;
d = 2*a;
if( delta > 0 ) {
    printf("ecuatia are 2 radacini reale si\ndistincte\n");
    delta = sqrt(delta);
    printf("x1=%g\tx2=%g\n",(-b + delta)/d,
        (-b - delta)/d);
    exit(0);
}
if( delta == 0 ) {
    printf("ecuatia are radacina dubla\n");
    printf("x=%g\n", -b/d);
    exit(0);
}
printf("radacini complexe\n");
delta = sqrt(-delta)/d;
d = -b/d;

```

```
printf("x1=%g+1(%g)\n",d,delta);
printf("x2=%g-1(%g)\n",d,delta);
}
```

## 4.6. Instrucțiunea while

Instrucțiunea *while* are formatul:

```
while(expresie)
    instrucțiune
```

Primul rând din acest format constituie antetul instrucțiunii *while*, iar *instrucțiune* este corpul ei.

La întâlnirea acestei instrucțiuni întâi se evaluează expresia din paranteze. Dacă ea are valoarea *adevărată* (este diferită de zero), atunci se execută *instrucțiune*. Apoi se revine la punctul în care se evaluează din nou valoarea expresiei din paranteze. În felul acesta, corpul ciclului se execută atît timp cît expresia din antetul ei este diferită de zero. În momentul în care *expresie* are valoarea zero, se trece la instrucțiunea următoare instrucțiunii *while*.

Corpul instrucțiunii *while* poate să nu se execute niciodată. Într-adevăr dacă *expresie* are valoarea zero de la început, atunci se trece la instrucțiunea următoare instrucțiunii *while* fără a executa niciodată corpul instrucțiunii respective.

Corpul instrucțiunii *while* este o singură instrucțiune, care poate fi compusă. În felul acesta, avem posibilitatea să executăm repetat mai multe instrucțiuni grupate într-o instrucțiune compusă.

Corpul instrucțiunii *while* poate fi o altă instrucțiune *while* sau să fie o instrucțiune compusă care să conțină instrucțiuni *while*. În acest caz se spune că instrucțiunile *while* respective sînt imbricate.

Menționăm că instrucțiunea din corpul unei instrucțiuni *while* poate să definească un alt mod de execuție a instrucțiunii *while* decît cel indicat mai sus. Astfel, ea poate realiza terminarea instrucțiunii *while* fără a se mai ajunge la evaluarea expresiei din antetul ei. De exemplu, dacă în corpul unei instrucțiuni *while* se apelează funcția *exit*, atunci se va termina execuția ciclului *while*, deoarece se intrerape chiar execuția programului.

Despre instrucțiunea *while* se spune adesea că este o instrucțiune *cică*.

Amintim că ea definește o structură repetitivă condiționată anterior.

### Exerciții:

4.9. Să se scrie un program care calculează și afișează valoarea polinomului

$$p(x)=3*x*x-7*x-10$$

pentru

$$x=1, 2, \dots, 10$$

Programul de față realizează un proces de calcul care constă din evaluarea și afișarea valorii expresiei:

$$3*x*x-7*x-10$$

pentru cele 10 valori ale lui  $x$ . Pentru o valoare a lui  $x$ , acest lucru se realizează prin intermediul instrucțiunii de apel:

```
printf("x=%d\tp(x)=%d\n",x,3*x*x-7*x-10);
```

Rezultă că, această instrucțiune trebuie să se execute repetat pentru  $x=1, 2, \dots, 10$ . În acest scop, inițial vom atribui lui  $x$  valoarea 1:

```
x=1;
```

Apoi, vom utiliza o instrucțiune *while* care să permită execuția repetată a instrucțiunii de apel a funcției *printf* de mai sus.

Deoarece lui  $x$  i s-a atribuit deja valoarea 1, rezultă că la prima execuție a acestei instrucțiuni se calculează  $p(1)$ . Pentru ca la a doua execuție a ei să se calculeze  $p(2)$ , este necesar să se mărească valoarea lui  $x$  cu 1. Aceasta se poate realiza folosind instrucțiunea expresie

```
x++;
```

Cu alte cuvinte, este suficient să se execute repetat secvența de instrucțiuni:

```
printf("x=%d\tp(x)=%d\n",x,3*x*x-7*x-10),  
x++;
```

$x$  avînd inițial valoarea 1. Această secvență formează corpul ciclului. De aceea, ea va fi inclusă între acolade pentru a forma o singură instrucțiune.

Deoarece ultima execuție a secvenței de mai sus are loc cînd  $x=10$ , rezultă că ea se va executa repetat atîta timp cît  $x \leq 10$ . De aceea, antetul ciclului *while* va fi:

```
while(x<=10)
```

În concluzie, procesul de calcul ce urmează a fi realizat îl putem descrie astfel:

1.  $x=1$
2. Atît timp cît  $x \leq 10$  se execută:

2.1. Afișează pe  $x$  și  $p(x)$ .

2.2. Incrementează pe  $x$ .

## PROGRAMUL BIV9

```
#include <stdio.h>
main() /* afiseaza valorile polinomului
         $p(x)=3*x*x - 7*x - 10$  pentru  $x = 1,2,...,10$  */
{
    int x;

    x = 1;
    while( x <= 10) {
        printf("x=%d\tp(x)=%d\n", x, 3*x*x - 7*x - 10);
        x++;
    } /* sfirsit while */
} /* sfirsit main */
```

### Observație:

Limbajul C permite scrierea de programe compacte. De exemplu, secvența de mai sus poate fi compactată înlocuind expresia

$x \leq 10$

cu

$++x \leq 10$

În felul acesta se elimină instrucțiunea expresie

$x++$ ;

din corpul ciclului.

Ținând seama de ordinea operațiilor, expresia

$++x \leq 10$

se evaluează astfel:

1. Se incrementează  $x$ .
2. Se compară valoarea incrementată a lui  $x$  cu 10.

Dacă aceasta nu-l depășește pe 10, atunci se execută corpul ciclului, care acum constă numai din apelul funcției *printf*.

În caz contrar se încheie execuția ciclului *while* și prin aceasta se termină

și execuția programului.

În acest caz se va schimba atribuirea

`x=1`

cu

`x=0`

deoarece execuția corpului ciclului începe după incrementarea lui `x`. Cu alte cuvinte, programul BIV9 poate fi rescris astfel:

```
...
x=0;
while(++x<=10)
    printf("x=%d\t p(x)=%d\n",x,3*x*x-7*x-10);
} /* sfirsit main */
```

Menționăm că expresia

`x++ <= 10`

utilizată în antetul ciclului nu realizează același lucru.

Într-adevăr, în cazul expresiei

`++x <= 10`

atât comparația, cât și corpul ciclului *while* se execută cu valoarea incrementată a lui `x`.

În cazul expresiei

`x++ <= 10`

comparația se face cu valoarea neincrementată a lui `x`, iar corpul ciclului *while* se execută cu valoarea incrementată a lui `x`. De aceea, dacă `x=10`, atunci

`++x <= 10`

este falsă, deci corpul ciclului nu se mai execută, în schimb

`x++ <= 10`

are valoarea adevărat și se va executa corpul ciclului și pentru `x=11`. De aceea, se va obține același lucru dacă în antetul ciclului vom folosi expresia

`x++<10`

4.10 Să se scrie un program care tablează valorile funcției  $\sin(x)$  cu pasul

de un grad sexagesimal,  $x$  aparținind intervalului  $[0,360)$ .

Funcția *sinus*, poate fi apelată ca funcție standard și are prototipul

**double sin(double x);**

în fișierul *math.h*.

Parametrul  $x$  este în radiani.

Dacă  $a$  este un unghi în grade sexagesimale, atunci el se reprezintă în radiani dacă se înmulțește cu factorul  $PI/180$  ( $PI=3.14159265\dots$ ):

**$a*PI/180$**

Tabelarea funcției *sinus* se va realiza conform pașilor de mai jos:

1.  **$f=PI/180$**
2.  **$x=0$**
3. Atât timp cât  $x \leq 359$ , se execută:
  - 3.1. Calculează și afișează  $\sin(x*f)$ .
  - 3.2. Incrementează pe  $x$ .

Pasul 3 de mai sus este analog cu pasul 2 din exemplul precedent. De aceea, el se realizează printr-un ciclu *while* asemănător. La realizarea lui vom face compactarea indicată în observația de la exemplul respectiv.

## PROGRAMUL BIV10

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

main() /* calculeaza si afiseaza valorile functiei sin(x) din grad
       in grad sexagesimal, x aparținind intervalului [0,359] */
{
    int x;
    double f;

    f = PI/180.0;
    x = -1;
    while(++x <= 359 )
        printf("sin(%d)=%.16f\n", x, sin(x*f));
}
```

### Observație:

Valorile funcției *sinus* se afișează câte una pe o linie. De aceea, programul afișează 360 de linii. Utilizatorul are posibilitatea de a vedea pe ecran, în fereastra utilizator (tastind <Alt>-F5), numai ultimele 25 de valori. Aceasta din cauză că un ecran, în mod text, se compune din 25 de linii a 80 de coloane.

În astfel de situații se pune problema blocării execuției unui program după ce acesta a afișat un ecran de rezultate. După analiza datelor de pe ecran, utilizatorul urmează să deblocheze execuția programului acționind o tastă oarecare.

Un mod simplu de a bloca execuția unui program este acela de a apela funcția *getch* în momentul în care se dorește să se analizeze datele afișate în fereastra utilizator.

La apelul funcției *getch* programul se blochează deoarece se așteaptă acționarea unei taste. Totodată se afișează automat fereastra utilizator. La acționarea unei taste oarecare, programul se deblochează.

În exemplul de față vom apela funcția *getch* ori de câte ori programul a afișat 23 de valori ale funcției *sinus*. Aceste valori ocupă 23 de linii ale ecranului. Pe linia 24 se va afișa textul:

pentru a continua acționați o tastă

Funcția *getch* se apelează în corpul lui *while* de îndată ce sînt afișate 23 de valori. În acest scop utilizăm valorile lui *x* care cresc de la 0 la 359. Ecranul trebuie blocat după apelul funcției *printf* și în momentul în care *x+1* are valorile:

23, 46, 69,...

adică atunci cînd *x+1* este multiplu de 23.

Aceasta se exprimă prin expresia:

$(x+1)\%23==0$

Deci funcția *getch* se apelează cînd expresia de mai sus este adevărată. Înainte de a apela funcția *getch* va trebui apelată funcția *printf* pentru a afișa, pe linia 24, textul:

pentru a continua acționați o tastă

Ținînd seama de aceste observații, modificăm programul de față ca mai jos.

## PROGRAMUL BIV10A

```
#include <stdio.h>
#include <math.h>
#include <conio.h>

#define PI 3.14159265358979

main() /* calculeaza si afiseaza valorile functiei sin(x) din grad
      in grad sexagesimal, x apartinind intervalului [0,359] */
{
    int x;
    double f;

    f = PI/180.0;
    x = -1;
    while(++x <= 359 ) {
        printf("sin(%d)=%.16f\n", x,sin(x*f));
        if((x+1)%23 == 0 ) {
            printf("pentru a continua actionati o\
                   tasta\n");
            getch();
        }
    }
    /* sfirsit while */
} /* sfirsit main*/
```

- 4.11 Să se scrie un program care citește un șir de întregi separați prin caractere albe și afișează suma lor. După ultimul număr se va tasta un caracter alb urmat de un caracter nenumeric (de exemplu sfârșitul de fișier: <Ctrl>-z, o literă etc), iar după acesta se va acționa tasta *Enter*.

Programul de față are o parte care se execută repetat.

- Citește construcția curentă de la intrare.
- Dacă la intrare s-a aflat un întreg, atunci acesta se adună la valoarea unei variabile *s* și se revine la punctul a.

Altfel se întrerupe citirea.

De aici rezultă că inițial *s* trebuie să aibă valoarea zero.

Un întreg se citește apelând funcția *scanf*. Dacă notăm cu *i* variabila căreia *i* se atribuie întregul citit, atunci la citirea unui întreg, expresia.

```
scanf("%d",&i)==1
```

are valoarea *adevărat*. Deci valoarea lui *i* trebuie adunată la *s* atît timp cit



expresia de mai sus este *adevărată*. Deci expresia de mai sus se utilizează în antetul instrucțiunii *while*, iar corpul ciclului se compune din instrucțiunea:

```
s=s+i;
```

## PROGRAMUL BIV11

```
#include <stdio.h>
main() /* citește un sir de intregi separati prin caractere
        albe si afiseaza suma lor */
{
    int i,s;

    s = 0;
    while(scanf("%d",&i) == 1)
        s += i;
    printf("suma = %d\n", s);
}
```

4.12 Să se scrie un program care citește un întreg  $n$ , calculează și afișează pe  $n!$ .

Avem:

$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$ , pentru  $n > 0$  și  $0!$  se consideră egal cu 1.

Acest calcul implică un proces ciclic.

Într-adevăr, fie

$f=1$  și  $i=2$

Atunci:

$f \cdot i = 1 \cdot 2 = 2!$

Considerăm instrucțiunea de atribuire:

$f=f \cdot i$ ;

Cum  $f \cdot i = 2!$ , rezultă că  $f = 2!$ . Putem folosi aceeași instrucțiune pentru a calcula pe  $3!$ .

Într-adevăr

$3! = 2! \cdot 3 = f \cdot 3$

deci expresia:  $f \cdot i = 3!$  dacă, în prealabil,  $i$  se mărește cu o unitate. Rezultă că alături de instrucțiunea de mai sus, este necesar să considerăm și instrucțiunea de incrementare a lui  $i$ :

- a.  $f=f*i$ ;
- b.  $i=i+1$ ;

Această secvență se execută repetat, întâi *a* apoi *b* și apoi din nou *a* și apoi *b* și așa mai departe. Lui *f* i se atribuie succesiv valorile:

2!, 3!, 4!,...

Pentru a calcula pe  $n!$  ( $n > 1$ ), secvența de instrucțiuni a-b trebuie să se execute repetat atât timp cât *i* nu-l depășește pe *n*, adică atât timp cât expresia

$i \leq n$

este adevărată.

Acest proces de calcul se realizează simplu prin instrucțiunea *while* de mai jos:

```
while (i<=n) {
    f=f*i;
    i++;
}
```

Instrucțiunea compusă poate fi compactată astfel.

```
f=f*i++;
```

Într-adevăr, operatorul de incrementare se aplică postfixat. Deci *f* se înmulțește cu valoarea neincrementată a lui *i*, ca și în cazul instrucțiunilor compuse inițiale.

## PROGRAMUL BIV12

```
#include <stdio.h>
#include <stdlib.h>

main() /* citește pe n din intervalul [1,170], calculează
        și afișează pe n! */
{
    int n,i;
    double f;

    printf("valoarea lui n:");
    if (scanf("%d",&n) != 1) {
        printf("nu s-a tastat un intreg\n");
        exit(1);
    }
    if ( n < 0 || n > 170 ) {
```

```

        printf("n nu apartine intervalului\
        [0,170]\n");
        exit(1);
    }
    f = 1.0;
    i = 2;
    while( i <= n )
        f = f*i++;
    printf("n=%d\tn!=%g\n",n,f);
}

```

- 4.13 Să se scrie un program care citește componentele a doi vectori  $x$  și  $y$ , calculează și afișează valoarea produsului lor scalar.

Dacă vectorul  $x$  are componentele:

$x_1, x_2, \dots, x_n$

iar  $y$ :

$y_1, y_2, \dots, y_n$

atunci produsul lor scalar se definește prin suma de produse:

$(x,y)=x_1*y_1+x_2*y_2+\dots+x_n*y_n$

Componentele celor doi vectori se tastează în următoarea ordine:

$x_1 \ y_1$

$x_2 \ y_2$

...

$x_n \ y_n$

Pe rândul care urmează după ultima pereche se poate tasta o literă, un caracter special care nu intră în compunerea unui număr sau sfârșitul de fișier.

### PROGRAMUL BIV13

```

#include <stdio.h>
main() /* citește componentele vectorilor x și y, calculează și
        afișează valoarea produsului lor scalar */
{
    double x,y,prodscal;
    int i;

```

```

prodscal = 0.0;
printf("componentele lui x si y\n");
i = 0;
printf("x[%d]\ty[%d]:", i+1,i+1);
while(scanf("%lf %lf", &x,&y) == 2) {
    prodscal += x*y;
    i++;
    printf("x[%d]\ty[%d]:", i+1,i+1);
}
printf("numarul componentelor vectorilor=%d\n",i);
printf("produsul scalar=%g\n",prodscal);
}

```

- 4.14 Să se scrie un program care citește fără ecou caractere imprimabile și le afișează ca și caractere minus. Programul se intrerupe la tastarea unui caracter care nu aparține codului ASCII.

La realizarea programului vom folosi funcțiile *getch* și *putch*.

Funcția *getch* returnează codul ASCII al caracterului citit sau zero dacă se citește un caracter care nu aparține codului ASCII. Expresia

```
c=getch()
```

atribuie variabilei *c* codul ASCII al caracterului citit. Se vor recodifica prin minus numai caracterele imprimabile, adică de cod ASCII cel puțin egal cu 32 și cel mult 126. Restul caracterelor se neglijează. Dacă *c* are ca valoare codul ASCII al caracterului citit, atunci instrucțiunea

```

if(c>=32&& c<=126)
    putch('~');

```

permite afișarea caracterului respectiv ca și caracter minus, dacă acesta este un caracter imprimabil; altfel el este neglijat. Această instrucțiune se execută repetat, cât timp se tastează un caracter care aparține codului ASCII. De aceea, ea va forma corpul unui ciclu *while*.

În antetul acestui ciclu vom folosi expresia

```
c=getch()
```

care este diferită de zero (adevărată) pentru caracterele codului ASCII și zero (falsă) pentru restul caracterelor.

## PROGRAMUL BIV14

```

#include <conio.h>
main() /* citește caractere imprimabile fara ecou si le afiseaza

```

```

        ca si caracter minus */
{
    int c;

    while ( c = getch() )
        if( c >= 32 && c <= 126 )
            putchar('-');
}

```

- 4.15 Să se scrie un program care apelează funcția *putch* cu valorile codului ASCII extins.

Codul ASCII extins cuprinde valorile întregi din intervalul [0,255]

Pe o linie se va afișa valoarea codului urmată de imaginea cores punzatoare afișată prin funcția *putch*. Ecranul se blochează după 23 de linii afișate sub forma indicată mai sus și după ce pe linia 24 se afișează mesajul:

pentru a continua actionati o tasta

După acționarea unei taste ecranul se deblochează pentru a se afișa un nou set de imagini și așa mai departe.

#### PROGRAMUL BIV15

```

#include <stdio.h>
#include <conio.h>

main() /* apeleaza functia putch cu valorile codului ASCII extins */
{
    int c;

    c = -1;
    while ( ++c <= 255 ) {
        printf("cod:%4d\t",c);
        putch(c);
        putchar('\n');
        if(( c+1) % 22 == 0) {
            printf("pentru a continua actionati o\
                    tasta\n");
            getch();
        }
    }
}

```

4.16 Să se scrie un program care citește rezultatele unor măsurători:

$x_1, x_2, \dots, x_n$

calculează și afișează:

→ media aritmetică a măsurătorilor;

→ media geometrică a lor;

și

→ abaterea pătratică.

Dacă notăm cu *meda* media aritmetică a măsurătorilor, atunci abaterea pătratică este rădăcina pătrată din expresia:

$$\{(x_1 - \text{meda})^2 + (x_2 - \text{meda})^2 + \dots + (x_n - \text{meda})^2\} / N$$

unde:

$N$  - Este  $n$  sau  $n-1$ .

Numitorul se ia egal cu  $n-1$  când  $n$  este relativ mic. De obicei:

$N = n-1$  pentru  $n < 30$

și

$N = n$  pentru  $n \geq 30$

Evident  $n \geq 2$ .

Deoarece pentru calculul abaterii pătratice este nevoie de măsurătorile  $x_1, x_2, \dots, x_n$ , cit și de media lor aritmetică, programul va trebui să păstreze măsurătorile respective într-un tablou unidimensional, pe care îl notăm cu  $x$ .

Vom presupune că există cel mult 500 de măsurători ( $n \leq 500$ ).

Media geometrică se calculează prin extragerea rădăcinii de ordinul  $ng$  din produsul celor  $ng$  măsurători pozitive dintre cele  $n$  măsurători citite.

Setul de măsurători sînt date flotante în simplă precizie. Setul de măsurători se consideră terminat dacă se tastează un caracter care nu intră în compunerea unei date numerice.

Programul listează măsurătorile citite, cite 5 pe un rînd. De asemenea, programul se blochează după afișarea a 23 de linii cu măsurători.

## PROGRAMUL BIV16

```
#define MAXMAS 500
```

```

#define LIMNUMIT 30
#define APEL printf("masuratoarea a %d-a:", n+1)

#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>

main() /* citeste un set de masuratori, calculeaza si afiseaza
        media lor aritmetica, geometrica si abaterea patratica */
{
    float x[MAXMAS];
    double meda,medg,ap;
    int n,ng,i;

    n=ng=0;
    meda=0.0;
    medg=1.0;
    APEL;

    /* - se citesc masuratorile;
       - se insumeaza;
       - se inmultesc cele pozitive
    */
    while(scanf("%f", &x[n] ) == 1) {
        meda += x[n];
        if( x[n] > 0 ) {
            medg *= x[n];
            ng++;
        }
        n++;
        APEL;
    }
    if( n < 1 ) {
        printf("nu s-a tastat nici o masuratoare\n");
        exit(1);
    }
    meda /= n;

    /* calculeaza media geometrica */
    if( ng < 2 ) {
        printf("nu exista nici 2 masuratori\
                pozitive\n");
    }
}

```

```

        medg = -1.0;
    }
    else
        medg = pow(medg, 1.0/ng);
    if( n > 1 ) {

/* calculeaza abaterea patratica */
        ap = 0.0;
        i = 0;
        while ( i < n ) {
            ap += (x[i] - meda)*(x[i] - meda);
            i++;
        }
        if( n < LIMNUMIT)
            ap /= n-1;
        else
            ap /= n;
    }
    else {
        printf("nu exista nici 2 masuratori\n");
        ap = -1;
    }
}

/* listeaza masuratorile, cite 5 pe un rind */
i = 0;
while(i < n) {
    printf("x[%d]=%g%c", i+1, x[i],
           i%5==4 || i==n-1 ? '\n':' ');
    i++;
    if(i%(5*23) == 0 ) {
        printf("pentru a continua actionati o
               tasta\n");
        getch();
    }
}

/* afiseaza mediile si abaterea patratica */
printf("media aritmetica=%g\n", meda);
if(medg != -1.0)
    printf("media geometrica=%g\n", medg);
if( n > 1)
    printf("abaterea patratica=%g\n", sqrt(ap));
}

```



### Observații:

1. Pentru a afișa măsurătorile cite 5 pe un rînd s-a apelat funcția *printf* folosind trei specificatori de format în parametrul de control:

`"x[%d] = %g%c"`

Parametrul corespunzător lui `%d` este `i+1`. Acesta definește numărul de ordine al măsurătorii curente.

Parametrul corespunzător lui `%g` este `x[i]` și reprezintă chiar valoarea măsurătorii curente.

După specificatorul `%g` urmează specificatorul `%c`, a cărui parametru definește caracterul care se afișează după valoarea măsurătorii curente. Acest caracter poate fi spațiu (`' '`) sau caracterul de rînd nou (`'\n'`).

Pe primul rînd se afișează valorile:

`x[0] x[1] x[2] x[3] x[4]`

deci, după măsurătorile de indice 0, 1, 2 și 3 se afișează un spațiu, iar după `x[4]` urmează caracterul de rînd nou.

Analog, pe rîndul al doilea se afișează valorile:

`x[5] x[6] x[7] x[8] x[9]`

deci, după măsurătorile de indice 5, 6, 7 și 8 se afișează un spațiu, iar după `x[9]` urmează caracterul de rînd nou.

În general, pe un rînd se afișează valorile.

`x[5k] x[5k+1] x[5k+2] x[5k+3] x[5k+4]`

Se observă că după o măsurătoare de indice `5k+4` urmează caracterul de rînd nou, iar în rest urmează un spațiu.

Un indice  $i$  este de forma  $5k+4$ , dacă restul împărțirii lui  $i$  la 5 este egal cu 4, adică, dacă expresia:

$$i \% 5 == 4$$

este adevărată.

De asemenea, după ultima măsurătoare se va afișa caracterul de rînd nou, adică dacă

$$i == n-1$$

are valoarea *adevărat*.

Rezultă că afișarea caracterului de rînd nou se va face atunci cînd expresia

`i%5==4 || i==n-1`

este adevărată. În rest se afișează un caracter spațiu. Această situație se reprezintă simplu prin expresia condițională:

`i%5==4 || i==n-1?"\n":' '`

expresie ce corespunde specificatorului de format `%c`.

Acest exemplu demonstrează odată în plus facilitățile mari oferite de expresiile condiționale la compactarea programelor.

Această metodă de afișare a fost indicată de autori limbajului C în lucrarea [2].

2. Pentru a calcula radicalul de ordinul  $ng$  din  $medg$  se apelează funcția `pow` cu parametrul  $medg$  și  $1.0/ng$ .

Funcția `pow` are prototipul:

`double pow (double x, double y);`

și returnează pe  $x$  la puterea  $y$ . În cazul de față  $1.0/ng$  are tipul `double` deoarece  $1.0$  este o constantă de tip `double`. Menționăm că expresia  $1/ng$  are valoarea zero (împărțire de întregi cu  $ng > 1$ ) deci este necesară utilizarea constantei  $1.0$ . Apelul funcției `pow` în programul de mai sus returnează pe  $medg$  la puterea  $1.0/ng$ , adică radical indice  $ng$  din  $medg$ .

- 4.17 Să se scrie un program care citește un cuvânt (succesiune de caractere diferite de caracterele albe) și afișează prefixele și sufixele lui diferite de el însuși.

Fie cuvîntul:

`e+t*f`

atunci prefixele acestui cuvînt, diferite de el însuși sînt:

`e`

`e+`

`e+t`

`e+t*`

În mod analog, sufixele aceluiași cuvînt sînt:

`f`

`*f`

`t*f`

`+t*f`

Se presupune că un cuvînt nu depășește 70 de caractere.

Programul avansează peste eventualele caractere albe care preced cuvîntul de prelucrat. Apoi citește caracterele din compunerea cuvîntului și le păstrează ca elemente ale tabloului *tab*.

Cuvîntul se termină la:

- întîlnirea unui caracter alb;
- întîlnirea sfîrșitului de fișier;
- după ce s-au citit 70 de caractere care nu sînt albe.

Caracterele se citesc folosind macroul *getchar*. Acesta se apelează prin expresia de atribuire:

```
c=getchar()
```

care atribuie lui *c* codul ASCII al caracterului citit. Pentru a avansa peste caracterele albe este necesar să stabilim dacă codul atribuit lui *c* este codul spațiului, al tabulatorului sau al caracterului de rînd nou. Expresia

```
(c=getchar())==' '
```

compară codul caracterului citit cu cel al spațiului și are valoarea *adevărat*, dacă s-a citit un spațiu și *fals* în caz contrar.

În mod analog, expresia

```
c=='\t'
```

este adevărată dacă s-a citit caracterul tabulator, iar expresia

```
c=='\n'
```

este adevărată dacă s-a citit caracterul *newline*. Rezultă că, dacă se citește un caracter alb, atunci este adevărată una dintre cele trei expresii de mai sus, ceea ce conduce la faptul că expresia:

```
(1) (c=getchar())==' ' || c=='\t' || c=='\n'
```

este adevărată dacă și numai dacă se citește un caracter alb.

De aici, rezultă că pentru a avansa peste caracterele albe vom utiliza un ciclu *while* care se va executa atît timp cit expresia (1) este adevărată. Deci antetul ciclului este:

```
while((c=getchar())==' ' || c=='\t' || c=='\n')
```

Deoarece acest ciclu nu are de realizat altceva decît să avanseze peste caracterele albe, corpul lui se compune din instrucțiunea vidă.

Citirea caracterelor cuvîntului de la intrare se realizează tot printr-o

instrucțiune *while*. De data aceasta, ciclul *while* se execută atita timp cit expresia (1) este falsă, nu s-a intulnit sfirșitul de fișier și nici nu s-au citit încă 70 de caractere.

Prima condiție se exprimă prin negația expresiei (1), presupunind că *c* conține deja caracterul curent.

(2)  $!(c == ' ' | c == '\t' | c == '\n')$

Ținind seama de faptul că expresia

$!(a | b)$

este echivalentă cu:

$!a \& \& !b$

expresia (2) se poate pune sub forma:

(3)  $!(c == ' ') \& \& !(c == '\t') \& \& !(c == '\n')$

Deoarece expresia:

$!(a == b)$

este echivalentă cu:

$a != b$

expresia (3) se poate pune sub o formă mai simplă:

(4)  $c != ' ' \& \& c != '\t' \& \& c != '\n'$

A doua condiție pentru continuarea ciclului *while* este ca să nu se citească sfirșitul de fișier. Aceasta înseamnă că expresia.

(5)  $cl = EOF$

trebuie să fie adevărată.

Dacă notăm cu *i* numărătorul de caractere citite, diferite de caracterele albe, atunci ciclul *while* continua dacă pe lângă condițiile (4) și (5) de mai sus, este adevărată și condiția exprimată prin expresia:

(6)  $i < 70$  (se presupune că inițial *i* a fost 0)

De aici rezultă că ciclul *while* pentru citirea caracterelor cuvintului aflat la intrare se continuă atit timp cit expresiile (4), (5) și (6) sint adevărate, ceea ce se exprimă conectind expresiile respective prin operatorul "și logic".

În corpul ciclului se păstrează codul caracterului citit în tabloul *t* de tip caracter, se incrementează numărătorul de caractere și apoi se citește caracterul următor. Acestea se realizează prin instrucțiunile de atribuire:

`t[i++] = c;`

și

`c = getchar();`

Pentru determinarea prefixelor cuvîntului urmăm pașii de mai jos

unde:

- i* - Are ca valoare lungimea cuvîntului.
- j* - Are ca valoare lungimea prefixului curent.
- k* - Numără caracterele unui prefix.

1.  $j = 1$  - primul prefix are lungimea 1.
2. Cît timp  $j < i$  se execută:
  - 2.1.  $k = 0$ .
  - 2.2. Cît timp  $k < j$  se execută:
    - 2.2.1. Se afișează `t[k]`.
    - 2.2.2. Se mărește  $k$  cu o unitate.
  - 2.3. Se afișează *newline* deoarece s-au afișat toate caracterele prefixului curent.
  - 2.4. Se mărește  $j$  cu o unitate pentru a afișa prefixul următor

Pașul 2 se realizează prin două cicluri *while* imbricate:

```
while(j < i){  
    k = 0; /* punctul 2.1. */  
    while(k < j){ /* punctul 2.2. */  
        putchar(t[k]); /* punctul 2.2.1. */  
        k++; /* punctul 2.2.2. */  
    }  
    putchar(' \n' ); /* punctul 2.3. */  
    j++; /* punctul 2.4. */  
}
```

Sufixele se pot determina printr-o secvență analogă. Dacă în cazul prefixelor acestea toate încep cu `t[0]`, în cazul sufixelor, caracterul de început este variabil. Astfel, sufixul de lungime 1 conține numai ultimul caracter al cuvîntului, deci este `t[i-1]`. Suffixul de lungime 2 se compune din ultimele 2 caractere ale cuvîntului:

`t[i-2]` și `t[i-1]`.

Sufixul de lungime 3 se compune din caracterele:

$t[i-3]$ ,  $t[i-2]$  și  $t[i-1]$ .

În general, sufixul de lungime  $j$  se compune din caracterele:

$t[i-j]$ ,  $t[i-j+1]$ , ...,  $t[i-1]$ .

De aceea, în acest caz variabila  $k$  nu se mai inițializează cu 0, ci cu  $i-j$ .

Deoarece toate sufixele au ca ultim caracter pe  $t[i-1]$ , rezultă că variabila  $k$  variază cu pasul 1 de la  $i-j$  până la  $i-1$ . În rest, secvența pentru determinarea sufixelor este aceeași cu cea pentru determinarea prefixelor.

## PROGRAMUL BIV17

```
#include <stdio.h>

#define MAX 70

#include <stdlib.h>

main() /* citește un cuvint și afișează prefixele și sufixele lui */
{
    int c,i,j,k;
    char t[MAX];

    /* avans peste eventualele caractere albe care preced cuvintul
       de prelucrat */
    while((c = getchar()) == ' ' || c == '\t' ||
           c == '\n')
        ;
    if( c == EOF ) {
        printf("cuvint vid\n");
        exit(1);
    }

    /* se citesc caracterele cuvintului de la intrare */
    i = 0; /* numărător de caractere */
    while(c != ' ' && c != '\t' && c != '\n'
           && c != EOF && i < 70) {
        t[i++] = c;
        c = getchar();
    }

    /* afișează prefixele cuvintului de lungime i */
```

```

j = 1; /* j are ca valoare lungimea prefixului curent */
while( j < i ) {
    k=0; /* numarator pentru caracterele prefixului: de la 0 la j-1 */
    while( k < j )
        putchar( t[k++] );
    putchar( '\n' ); /* dupa prefixul afisat */
    j++;
} /* sfirsit afisare prefixe */

/* afiseaza sufixele cuvintului citit */
j = 1; /* lungimea sufixului curent */
while( j < i ) {
    k = i-j; /* numarator pentru caracterele sufixului:
               de la i-j la i-1 */
    while( k < i )
        putchar( t[k++] );
    putchar( '\n' );
    j++;
}
}

```

## 4.7. Instrucțiunea for

Instrucțiunea *for*, ca și instrucțiunea *while*, se utilizează pentru a realiza o structură *repetitivă condiționată anterior*.

Formatul ei este:

```

for(exp1;exp2;exp3)
    instrucțiune

```

unde:

*exp1*, *exp2* și *exp3* - Sint expresii.

Antetul instrucțiunii *for* este:

```
for(exp1;exp2;exp3)
```

iar *instrucțiune* este corpul ei și ea se execută repetat.

Expresia *exp1* se numește partea de *inițializare* a ciclului *for*, iar *exp3* este partea de *reinițializare* a lui. Expresia *exp2* este condiția de continuare a ciclului *for* și ea joacă același rol cu expresia din ciclul *while*.

Instrucțiunea *for* se execută astfel:

1. Se execută secvența de inițializare definită de *exp1*.
2. Se evaluează expresia *exp2*.  
Dacă are o valoare diferită de zero (este *adevărată*), atunci se execută instrucțiunea care formează corpul ciclului.  
Altfel (expresia are valoarea zero adică *fals*) se termină execuția instrucțiunii *for* și se trece la instrucțiunea următoare.
3. După executarea corpului ciclului se execută secvența de reinițializare definită de *exp3*.

Apoi se reia execuția de la pasul 2.

Ca și în cazul instrucțiunii *while*, instrucțiunea din corpul ciclului *for* nu se execută niciodată dacă *exp2* are valoarea zero chiar de la început.

Expresiile din antetul lui *for* pot fi și vide. Caracterele punct și virgulă vor fi totdeauna prezente.

**Exemplu:**

Secvența de însumare a elementelor tabloului *tab*:

$s = \text{tab}[0] + \text{tab}[1] + \dots + \text{tab}[n-1]$

se realizează executând repetat instrucțiunea compusă:

```
{  
    s=s+tab[i];  
    i++;  
}
```

unde inițial  $s=0$  și  $i=0$ , atit timp cit  $i < n$ . Ea se poate realiza cu ajutorul instrucțiunii *for* de mai jos:

```
for (s=0, i=0; i<n; i++)  
    s=s+tab[i];
```

În acest caz expresia *exp1* este formată din două atribuiri:

$s=0, i=0$

*exp2* este reprezentată de condiția

$i < n$

iar *exp3* este expresia

$i++$



Conform definiției instrucțiunii *for*, la început se evaluează *exp1*, deci se execută atribuiriile

```
s=0
```

și

```
i=0
```

Se evaluează *exp2*, adică se determină valoarea condiției

```
i<n
```

Dacă ea este adevărată, atunci se execută corpul instrucțiunii *for*, adică suma

```
s=s+tab[i]
```

Apoi se evaluează *exp3*, adică se execută incrementarea

```
i++;
```

După incrementarea lui *i* se revine la evaluarea condiției *exp2*. În felul acesta, ciclul continuă până când condiția *i<n* devine falsă, adică în momentul în care *i* devine egal cu *n*. Deci ciclul se întrerupe după ce elementele:

```
tab[0], tab[1], ..., tab[n-1]
```

au fost adăugate la *s*.

Secvența de mai sus poate fi scrisă cu ajutorul instrucțiunii *while* astfel:

```
s=0;  
i=0;  
while(i<n){  
    s=s+tab[i];  
    i++;  
}
```

În general, instrucțiunea *for*:

```
for(exp1;exp2;exp3)  
    instrucțiune
```

poate fi scrisă cu ajutorul unei secvențe în care se utilizează instrucțiunea *while*:

```
exp1;  
while(exp2)  
{  
    instrucțiune
```

```
exp3;  
}
```

Această echivalare nu are loc într-un singur caz și anume atunci când, în corpul instrucțiunii se utilizează instrucțiunea *continue* (vezi mai jos).

Reciproc, orice instrucțiune *while* poate fi scrisă cu ajutorul unei instrucțiuni *for* în care *exp1* și *exp3* sînt vide.

Astfel, instrucțiunea *while* de mai jos:

```
while(exp)  
    instrucțiune
```

este echivalentă cu instrucțiunea *for*

```
for(;exp;)  
    instrucțiune
```

O instrucțiune *for* de forma

```
for(;;)  
    instrucțiune
```

este validă și ea este echivalentă cu instrucțiunea *while* de mai jos:

```
while(1)  
    instrucțiune
```

Un astfel de ciclu se poate termina prin alte mijloace decît cel obișnuit, cum ar fi instrucțiunea de revenire dintr-o funcție, un salt la o etichetă etc. (vezi mai jos).

Din cele de mai sus rezultă echivalența celor două cicluri. Autorii recomandă utilizarea instrucțiunii *for* în ciclurile în care sînt prezente părțile de inițializare și reinițializare. De obicei, aceste cicluri sînt așa numitele cicluri cu *pas*.

**Exerciții:**

4.18 Să se scrie un program care citește întregul *n* din intervalul [0,170], calculează și afișează pe *n!* .

Acest exercițiu a fost rezolvat cu ajutorul instrucțiunii *while*. Propunem cititorului să rescrie exercițiile rezolvate cu ajutorul instrucțiunii *while* înlocuind-o pe aceasta cu *for*.

## PROGRAMUL BIV18

```
#include <stdio.h>  
#include <stdlib.h>
```

```

main() /* citește pe n din intervalul [0,170], calculează
       și afișează pe n! */
{
    int n,i;
    double f;

    printf("valoarea lui n:");
    if(scanf("%d",&n) != 1) {
        printf("nu s-a tastat un intreg\n");
        exit(1);
    }
    if( n < 0 || n > 170 ) {
        printf("n nu apartine intervalului\
              [0,170]\n");
        exit(1);
    }
    for( f=1.0, i=2; i <= n; i++)
        f *= i;
    printf("n =%d\t n! =%g\n", n,f);
}

```

- 4.19 Să se scrie un program care citește pe  $n$  și  $a$ , calculează și afișează valoarea lui  $a$  la puterea a  $n$ -a ( $a^n$ ).  $n$  este întreg nenegativ, iar  $a$  este un număr oarecare. Calculele se fac în flotantă *long double*.

O metodă simplă pentru a ridica pe  $a$  la puterea a  $n$ -a, pentru  $n$  natural, este de a înmulți pe  $a$  cu el însuși de  $n$  ori

#### PROGRAMUL BIV19

```

#include <stdio.h>
#include <stdlib.h>

main() /* citește pe n și a, calculează și afișează pe a^n */
{
    int n,i;
    long double a,p;

    printf("Baza a = ");
    if(scanf("%Lf", &a) != 1) {
        printf("nu s-a tastat un numar\n");
        exit(1);
    }
    printf("Exponentul n = ");

```

```

if( scanf("%d", &n) != 1 || n<0 ) {
    printf("nu s-a tastat un intreg nenegativ\n");
    exit(1);
}
for(i=0,p=1.0L; i < n ; i++ )
    p *= a;
printf("a=%Lg\tn=%d\ta**n=%Lg\n",a,n,p );
}

```

#### Observații:

1. Funcția *pow*, de prototip:

**double pow(double x,double y);**

se utilizează pentru date de tip *double*.

2. Metoda utilizată în programul de față nu este eficientă pentru *n* relativ mare, deoarece necesită multe înmulțiri. Numărul lor poate fi redus substanțial procedind ca mai jos.

Fie

$f \leftarrow a$

Executind repetat instrucțiunea de atribuire

$f \leftarrow f * f$ ;

se generează puterile:

(1)  $a^{**2}, a^{**4}, a^{**8}, a^{**16}, a^{**32}, \dots$

adică exponenții lui *a* sînt puteri ale lui 2.

Termenii șirului (1) se generează printr-un număr relativ mic de operații de înmulțire.  $a^{**n}$  se poate exprima folosind numai factori din șirul (1) și eventual și pe *a*, dacă *n* este impar. Acest lucru rezultă din faptul că orice număr natural pozitiv, se poate exprima ca o sumă de puteri ale lui 2. Într-adevar, nu avem decît să reprezentăm numărul *n* în binar și să însumăm puterile lui doi din șirul:

(2) 1,2,4,8,16,32,64,...

care corespund cifrelor 1 din această reprezentare.

Correspondența dintre termenii șirului (2) și cifrele reprezentării binare se face de la dreapta spre stînga adică 1 se pune în corespondența cu ultima cifră a reprezentării binare, 2 cu penultima, 4 cu antepenultima etc.

### Exemplu:

Fie  $n=43$ . În binar  $n$  se reprezintă astfel.

$$n=101011$$

Rezultă că  $n$  este suma termenilor 1, 2, 8 și 32 aleși din șirul (2) conform cifrelor 1 din reprezentarea binară a lui:

$$n=1+2+8+32=43$$

Pentru a calcula pe  $a^{**}n$  va fi suficient să realizăm produsul factorilor  $a^{**}2$ ,  $a^{**}8$ , și  $a^{**}32$ , factori aleși din șirul

(1) și  $a$ .

Din cele de mai sus rezultă că pentru a reduce numărul înmulțirilor la calculul lui  $a^{**}n$  este suficient să generăm factorii șirului (1) și să-i alegem pe cei ai căror exponenși însumai ne dau pe  $n$ . Selectarea lor se poate face simplu pornind de la reprezentarea binară a lui  $n$ . Astfel, dacă ultima cifră binară a lui  $n$  este 1, atunci se selectează ca prim factor chiar  $a$ . Apoi, făcând o deplasare spre dreapta a lui  $n$  cu o poziție binară, se alege sau nu factorul  $a^{**}2$ , după cum ultima cifră binară a lui  $n$ , după ce s-a făcut deplasarea, este 1 sau nu. Procesul continuă până când  $n$  devine egal cu zero.

Procesul de calcul se definește astfel:

1.  $p=1$
2.  $f=a$
3. Cât timp  $n$  este diferit de zero se execută:
  - 3.1. Dacă ultima cifră binară a lui  $n$  este egală cu 1, atunci  $p=p*f$
  - 3.2. Se generează termenul următor din șirul (1):
$$f=f*f$$
- 3.3.  $n$  se deplasează spre dreapta cu o poziție binară.

Se observă că punctul 3.2. este necesar numai dacă  $n>1$ . Într-adevăr, dacă  $n=1$  atunci procesul de calcul se termină, deoarece prin deplasarea lui  $n$  la dreapta cu o poziție binară acesta devine egal cu zero și deci ciclul nu se mai reia. Având în vedere faptul că termenii șirului (1) cresc rapid, este important ca să suprimăm termenul care s-ar genera pentru  $n=1$ , deoarece cu toate că el nu este necesar, calculul lui poate conduce la o depășire flotantă superioară. În acest fel, punctul 3.2. se schimbă cu

dacă  $n>1$  atunci

$$f=f*f$$

Programul BIV19A ridică pe  $a$  la  $n$  folosind procesul de calcul descris mai sus.

## PROGRAMUL BIV19A

```
#include <stdio.h>
#include <stdlib.h>

main() /* citește pe n și a, calculează și afișează pe  $a^{**}n$  */
{
    int n,i;
    long double a,p,f;

    printf("Baza a = ");
    if(scanf("%Lf", &a) != 1) {
        printf("nu s-a tastat un numar\n");
        exit(1);
    }
    printf("Exponentul n = ");
    if(scanf("%d", &n) != 1 || n<0) {
        printf("nu s-a tastat un intreg nenegativ\n");
        exit(1);
    }
    i = n;
    for(p=1.0L,f = a; n ; n >>= 1) {
        if( n & 1 )
            p *= f; /* factor selectat din sirul (1) */
        if( n > 1 )
            f *= f; /* generează termenul următor din sirul (1) */
    }
    printf("a=%Lg\tn=%d\ta**n=%Lg\n",a,i,p );
}
```

### Observații:

1. *exp1* realizează inițializările de la punctele 1 și 2 din descrierea de mai sus  
 $p=1.0L, f=a$
2. *exp2*, condiția de continuare a ciclului s-a redus la  $n$ ; deci ciclul continuă atît timp cît  $n$  este adevărat, adică este diferit de zero.
3. Punctul 3.3 este partea de reinițializare a ciclului *for* și ea constă din expresia

$n >= 1$

care deplasează pe  $n$  cu un ordin binar spre dreapta.

#### 4. Expresia

$n \& 1$

are valoarea 1, dacă ultimul ordin binar al lui  $n$  este egal cu 1. În acest caz factorul generat în variabila  $f$  se selectează, deci se înmulțește cu  $p$ .

## 4.8. Instrucțiunea do-while

Instrucțiunea *do-while* realizează structura *cică condiționată posterior*. Această instrucțiune poate fi realizată cu ajutorul celorlalte instrucțiuni definite până în prezent. Cu toate acestea, prezența ei în limbaj mărește flexibilitatea în programare.

Ea are formatul:

```
do  
  instrucțiune  
while(expresie);
```

Instrucțiunea *do-while* se execută în felul următor:

1. Se execută *instrucțiune*
2. Se evaluează *expresie*; dacă aceasta are o valoare diferită de zero, atunci se revine la punctul 1, pentru a executa din nou *instrucțiune*; altfel (*expresia* are valoarea zero) se trece în secvență, adică la instrucțiunea următoare instrucțiunii *do-while*.

Se observă că în cazul acestei instrucțiuni întâi se execută *instrucțiune* și apoi se testează condiția de repetare a execuției ei.

Instrucțiunea *do-while* este echivalentă cu secvența:

```
instrucțiune  
while(expresie)  
  instrucțiune
```

Considerăm și în acest caz că, *instrucțiune* reprezintă corpul ciclului *do-while*.

În cazul instrucțiunii *do-while* corpul ciclului se execută cel puțin o dată, spre deosebire de cazul instrucțiunilor *while* și *for*, când este posibil să nu se execute niciodată

### Exerciții:

4.20 Să se scrie un program care citește un număr nenegativ, subunitar, calculează și afișează rădăcina pătrată din numărul respectiv cu o eroare mai mică decât 0,0000000001.

Programul de față nu folosește funcția *sqrt* din biblioteca standard.

Pentru a extrage rădăcina pătrată dintr-un număr  $a$  din intervalul  $(0,1)$ , se poate folosi metoda iterativă a lui Newton de mai jos.

Fie șirul:

$x[0], x[1], \dots, x[n], \dots$

unde:

(1)  $x[n+1] = 0,5(x[n] + a/x[n])$  pentru  $n=0,1,2,\dots$

Se demonstrează că șirul de mai sus este convergent și are limita egală cu rădăcina pătrată din  $a$ . Convergența este foarte rapidă pentru  $a$  subunitar. În acest caz se poate lua  $x[0] = 1$ .

Pentru a obține rădăcina pătrată cu precizia cerută, este suficient ca diferența, în valoare absolută, dintre doi termeni consecutivi ai șirului să fie mai mică decât

$EPS = 0.0000000001$

Să observăm că pentru a rezolva problema cu metoda indicată mai sus, la fiecare pas al calculului, sînt necesari numai doi termeni ai șirului:

a. Din  $x[n]$  se determină  $x[n+1]$  cu relația (1).

b. Se compară

$abs(x[n] - x[n+1])$  cu  $EPS$ ;

Dacă valoarea absolută respectivă nu este mai mică decât  $EPS$ , atunci se calculează  $x[n+2]$  folosind relația (1), în care  $x[n]$  se înlocuiește cu  $x[n+1]$ .

Deci se poate reveni la punctul a de mai sus, după ce lui  $x[n]$  i se atribuie valoarea  $x[n+1]$ .

Deci în loc să utilizăm tabloul  $x$ , este suficient să folosim două variabile  $x1$  și  $x2$  care păstrează în fiecare moment doi termeni consecutivi ai șirului de mai sus. Termenii șirului se obțin executînd repetat secvența de atribuiri:

$x1 = x2;$

$x2 = 0.5 * (x1 + a/x1);$



După fiecare execuție a acestei secvențe,  $x_1$  și  $x_2$  au ca valori doi termeni consecutivi ai șirului căutat.

Secvența se va repeta atît timp cît  $(2) \text{ abs}(x_1 - x_2) \geq \text{EPS}$ , test care îl facem după fiecare execuție a secvenței de atribuire. Dacă el se confirmă, atunci se repetă secvența respectivă, altfel valoarea variabilei  $x_2$  aproximează rădăcina pătrată din  $a$  cu o eroare mai mică decît EPS.

Acest proces de calcul se realizează imediat printr-o instrucțiune *do-while*.

Secvența de atribuire este corpul ciclului, iar expresia (2) reprezintă condiția de continuare a lui. Întrucît primul termen al șirului are valoarea 1, instrucțiunea *do-while* este precedată de atribuirea:

```
x2=1;
```

## PROGRAMUL BIV20

```
#include <stdio.h>
#include <stdlib.h>

#define EPS 1e-10

main() /* citește pe a din intervalul [0,1], calculează și
        afișează rădăcina pătrată din a */
{
    double a, x1, x2, y;

    printf("Tastati valoarea lui a = ");
    if(scanf("%lf", &a) != 1 || a < 0 || a > 1) {
        printf("Nu s-a tastat un numar in\
                intervalul [0,1]\n");
        exit(1);
    }
    if( a == 0 )
        x2 = 0; /* rădăcina pătrată din zero este zero */
    else
        if( a == 1 )
            x2 = 1; /* rădăcina pătrată din 1 este 1 */
        else { /* a este diferit de 0 sau 1 */
            x2 = 1;
            do {
                x1 = x2;
                x2 = 0.5*(x1 + a/x1);
```

```

        if( ( y = x1 - x2) < 0 )
            y = -y;
/* y = abs(x1-x2)*/

    } while( y >= EPS );
} /* sfirsit else */
printf("a=%11g\tsqrt(a)=%11g\n", a, x2);
}

```

#### Observații:

1. Instrucțiunea *do-while* poate fi scrisă mai compact astfel:

```

do {
    x1=x2;
    x2=0.5*(x1+a/x1);
}while( (y=x1-x2)<0?-y:y,>=EPS);

```

2. Pentru numere supraunitare se poate aplica aceeași metodă. În acest caz se poate lua ca prim termen al șirului chiar numărul din care se extrage rădăcina pătrată. O altă posibilitate este de a extrage rădăcina pătrată din inversul numărului. Aceasta deoarece metoda este rapid convergentă pentru numere subunitare.

Fie  $a > 1$ . Atunci  $x = 1/a$ . Dacă  $y$  este rădăcina pătrată din  $x$ , atunci  $1/y$  este rădăcina pătrată din  $a$ .

- 4.21 Să se scrie un program care citește un întreg pozitiv ce reprezintă o sumă exprimată în lei. Se cere să se afișeze numărul minim de bancnote și monede de 1000 lei, 500 lei, 200 lei etc. necesare pentru a exprima suma respectivă.

Procesul de calcul începe cu determinarea numărului de bancnote de 1000 de lei. În acest scop se împarte suma respectivă  $s$  la 1000:

$s/1000$

Pentru a determina numărul bancnotelor de 500 de lei se determină restul împărțirii lui  $s$  la 1000 și acest rest se împarte la 500 și așa mai departe. Se obțin pașii următori:

1.  $q = s/1000$ ; afișează  $q$
2.  $s = s \% 1000$ ;
3.  $q = s/500$ ; afișează  $q$

4.  $s = s \% 500;$
5.  $q = s / 200;$  *afișează q*
6.  $s = s \% 200;$
7.  $q = s / 100;$  *afișează q*
8.  $s = s \% 100;$
9.  $q = s / 50;$  *afișează q*
10.  $s = s \% 50;$
11.  $q = s / 20;$  *afișează q*
12.  $s = s \% 20;$
13.  $q = s / 10;$  *afișează q*
14.  $s = s \% 10;$
15.  $q = s / 5;$  *afișează q*
16.  $s = s \% 5;$
17.  $q = s / 1;$  *afișează q*
18.  $s = s \% 1.$

Se observă că secvența:

$q = s / n;$  *afișează q*  
 $s = s \% n;$

se repetă pentru valorile lui  $n$  egale cu

1000, 500, 200, 100, 50, 20, 10, 5 și 1

Pentru a putea executa repetat această secvență este nevoie de o metodă simplă prin care să se atribuie lui  $n$  valorile corespunzătoare. Aceasta se poate realiza dacă păstrăm datele respective într-un tablou de tip *int*. Fie:

$\text{mon}[0] = 1,$	$\text{mon}[1] = 5,$	$\text{mon}[2] = 10,$
$\text{mon}[3] = 20,$	$\text{mon}[4] = 50,$	$\text{mon}[5] = 100,$
$\text{mon}[6] = 200,$	$\text{mon}[7] = 500,$	$\text{mon}[8] = 1000.$

Atunci  $n$  poate fi înlocuit prin  $\text{mon}[i]$ , unde inițial  $i = 8$  și se decrementează la fiecare pas al ciclului. Deci urmează a se executa repetat secvența:

$q = s / \text{mon}[i];$  *afișează q*  
 $s = s \% \text{mon}[i];$   
 $i = i - 1;$

Această secvență se va executa repetat până când  $s$  devine zero.

## PROGRAMUL BIV21

```
#include <stdio.h>
#include <stdlib.h>

main() /* exprima o suma de lei in bancnote si monede de
        1000 lei, 500 lei etc. */
{
    int mon[9];
    long s,q;
    int i;

    printf("Tastati suma: ");
    if(scanf("%ld",&s) != 1 || s <= 0 ) {
        printf("nu s-a tastat un intreg pozitiv\n");
        exit(1);
    }
    mon[0] = 1; mon[1] = 5; mon[2] = 10; mon[3] = 20;
    mon[4] = 50; mon[5] = 100; mon[6] = 200;
    mon[7] = 500; mon[8] = 1000;
    i = 8;
    do {
        q = s/mon[i];
        if(q) /* afisarea se face daca q este diferit de zero */
            if( i < 6 ) /* monede */
                if( q == 1 ) /* exista o singura moneda */
                    if( i == 0 ) /* o moneda de 1 leu */
                        printf("o moneda de 1 leu\n");
                    else
                        printf("o moneda a %d lei\n",
                               mon[i]);
                else /* mai multe monede */
                    if( i == 0 ) /* monede de 1 leu */
                        printf("%ld monede de 1 leu\n",q);
                    else /* monede a mon[i] lei */
                        printf("%ld monede a %d lei\n",
                               q, mon[i]);
            else /* bancnote */
                if( q == 1 ) /* o singura bancnota */
                    printf("o bancnota a %d lei\n",mon[i]);
                else /* mai multe bancnote */
                    printf("%ld bancnote a %d lei\n",q,
                           mon[i]);
    }
```

```

    } while ( s != mon[i--]);
}

```

## 4.9. Instrucțiunea *continue*

Această instrucțiune se poate utiliza numai în corpul unui ciclu. Ea permite abandonarea iterației curente. Formatul ei este:

***continue;***

Efectul instrucțiunii *continue* este următorul:

- a. În corpul instrucțiunilor *while* și *do-while*:

La întâlnirea instrucțiunii *continue* se abandonează iterația curentă și se trece la evaluarea expresiei care stabilește continuarea sau terminarea ciclului respectiv (expresia inclusă între paranteze rotunde și care urmează după cuvântul cheie *while*).

- b. În corpul instrucțiunii *for*:

La întâlnirea instrucțiunii *continue* se abandonează iterația curentă și se trece la execuția pasului de reinițializare.

Instrucțiunea *continue* nu este o instrucțiune obligatorie. Prezența ei mărește flexibilitatea în scrierea programelor C. Ea conduce adesea la diminuarea nivelelor de imbricare ale instrucțiunilor *if* utilizate în corpul ciclurilor.

## 4.10. Funcțiile standard *sscanf* și *sprintf*

Biblioteca standard a limbajelor C și C++ conține funcțiile *sscanf* și *sprintf* care sînt analoge funcțiilor *scanf* și respectiv *printf*. Ele au un parametru în plus la apel și anume primul lor parametru este adresa unei zone de memorie în care se pot păstra caractere ale codului ASCII. Ceilalți parametri sînt identici cu cei întâlniți în corespondentele lor, *scanf* și respectiv *printf*.

Primul parametru al acestor funcții poate fi numele unui tablou de tip *char*, deoarece un astfel de nume are ca valoare chiar adresa de început a zonei de memorie care îi este alocată.

Funcția *sprintf* se folosește, ca și funcția *printf*, pentru a realiza conversii ale datelor de diferite tipuri din formatele lor interne, în formate externe reprezentate prin succesiuni de caractere. Diferența constă în aceea că, de data aceasta caracterele respective nu se afișează pe terminalul standard, ci se păstrează în zona de memorie definită de primul parametru al funcției

*sprintf*. Ele se păstrează sub forma unui șir de caractere și pot fi afișate ulterior din zona respectivă cu ajutorul funcției *puts*. De aceea, un apel al funcției *printf* poate fi totdeauna înlocuit cu un apel al funcției *sprintf*, urmat de un apel al funcției *puts*. O astfel de înlocuire este utilă când dorim să afișăm de mai multe ori aceleași date. În acest caz se apelează funcția *sprintf* o singură dată pentru a face conversiile necesare din format intern în format extern, rezultatele conversiilor păstrându-se într-un tablou de tip caracter. În continuare se pot afișa datele respective apelând funcția *puts* ori de câte ori este necesară afișarea lor. Funcția *sprintf*, ca și funcția *printf* returnează numărul octeților șirului de caractere rezultat în urma conversiilor efectuate.

#### Exemplu:

```
int zi,luna,an;
char data_calend[11];
...
sprintf(data_calend,"%02d/%02d/%d",zi,luna,an);
puts(data_calend);
...
puts(data_calend);
```

Funcția *sscanf* realizează, ca și funcția *scanf*, conversii din format extern în format intern. Deosebirca constă în faptul că de data aceasta caracterele nu sînt citite din zona tampon corespunzătoare tastaturii, ci ele provin dintr-o zonă de memorie a cărei adresă este definită de primul parametru al funcției *sscanf*. Aceste caractere pot proveni în zona respectivă în urma apelului funcției *gets*. În felul acesta, apelul funcției *scanf* poate fi înlocuit prin apelul funcției *gets* urmat de apelul funcției *sscanf*. Astfel de înlocuiri sînt utile cînd dorim să eliminăm eventualele erori apărute la tastarea datelor.

Funcția *sscanf*, ca și funcția *scanf*, returnează numărul cîmpurilor convertite corect conform specificatorilor de format prezenți în parametrul de control. La întîlnirea unei erori, ambele funcții își întrerup execuția și se revine din ele cu numărul cîmpurilor tratate corect. Analizînd valoarea returnată, se poate stabili dacă au fost prelucrate corect toate cîmpurile sau a survenit o eroare.

În caz de eroare se poate reveni pentru a introduce corect datele respective. În acest scop este necesar să se elimine caracterele începînd cu cel din poziția eronată.

În cazul în care se utilizează secvența:

```
gets
sscanf
```

abandonarea caracterelor respective se face automat reapelindu-se funcția *gets*. În cazul utilizării funcției *scanf* este necesar să se avanseze pînă la caracterul *newline* aflat în zona tampon atașată tastaturii sau să se vizioneze zona respectivă prin funcții speciale.

În exercițiile care urmează vom folosi secvențele formate din apelurile funcției *gets* urmate de apelurile lui *scanf*. O astfel de secvență se apelează repetat în cazul în care se întîlnesc erori în datele de intrare.

**Exemplu:**

```
char tab[255];
int zi,luna,an;
...
gets(tab);
scanf(tab,"%d %d %d",&zi,&luna,&an),
```

Amintim că funcția *gets* returnează valoarea *NULL* la întîlnirea sfîrșitului de fișier.

Funcțiile *scanf* și *sprintf* au prototipurile în fișierul *stdio.h*.

**Exerciții:**

4.22 Să se scrie un program care citește un întreg pozitiv de tip *long*, stabilește dacă acesta este prim și afișează un mesaj corespunzător.

## PROGRAMUL BIV22

```
#include <stdio.h>
#include <stdlib.h>
```

```
main() /* citește un întreg pozitiv de tip long, stabilește dacă
        acesta este un număr prim și afișează un mesaj
        corespunzător */
```

```
{
    long n;
    long i;
    int j;
    char tab[255];

    do {
        printf("tastati un intreg pozitiv:");
        if(gets(tab) == NULL) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
    }
```

```

    if(sscanf(tab,"%ld",&n) != 1 || n <= 0 ) {
        printf("nu s-a tastat un intreg\
            pozitiv\n");
        j = 1;
        continue; /* se va relua ciclul deoarece se trece la
                    evaluarea expresiei j care este diferita
                    de zero */
    }
    j = 0; /* ciclul se intrerupe deoarece s a citit corect
            un intreg pozitiv */
} while (j);
for( j = 1, i = 2; i*i <= n && j; i++)
    if(n%i == 0) /* numarul nu este prim */
        j = 0;
printf("numarul: %ld", n);
if( j == 0 )
    printf(" nu ");
printf(" este prim\n");
}

```

#### Observații:

1. Utilizarea instrucțiunii *continue* se poate omite folosind o instrucțiune *if* cu alternativă *else*:

```

if(sscanf(...) != 1 || n<=0){
    ...
    j=1;
}else
    j=0;

```

2. Ciclul *for* continuă atit timp cit expresia:

$i*i \leq n \&\& j$

este adevărată. Această expresie se evaluează de la stinga spre dreapta și din această cauză expresia

$i*i \leq n$

se evaluează și atunci cind  $j = 0$ . De aceea expresia respectivă este mai eficientă sub forma.

$j \&\& i*i \leq n$

În acest caz, pentru  $j = 0$  nu se mai evaluează restul expresiei.



Expresia:

$i * i \leq n$

rezultă din faptul că, pentru a stabili că un număr este prim, este suficient să încercăm divizibilitatea lui prin numere al căror pătrat nu-l depășește pe  $n$ .

O altă posibilitate este aceea de a înlocui expresia

$i * i \leq n$

cu

$i \leq \text{sqrt}(n)$

În acest caz este util să se calculeze rădăcina pătrată în afara ciclului *for*:

```
...
long q;
...
q=sqrt((double)n);
for(j=1, i=2; j&& i<=q; i++)
    if(n%i==0)
        j=0;
...
```

O altă simplificare posibilă este schimbarea instrucțiunii *if* din corpul ciclului *for* cu:

$j=n\%i$ ;

## 4.11. Instrucțiunea *break*

Instrucțiunea *break* este înrudită cu instrucțiunea *continue*. Ea are formatul:

**break;**

Poate fi utilizată în corpul unui ciclu. În acest caz, la întâlnirea instrucțiunii *break* se termină execuția ciclului în al cărui corp este inclusă și execuția continuă cu instrucțiunea următoare instrucțiunii ciclice respective.

Instrucțiunea *break*, la fel ca și instrucțiunea *continue*, mărește flexibilitatea la scrierea programelor în limbajele C și C++.

**Exerciții:**

- 4.23 Să se scrie un program care citește măsurile  $a$ ,  $b$ ,  $c$  ale laturilor unui triunghi, calculează și afișează aria triunghiului respectiv.

Pentru determinarea ariei se va folosi formula lui Heron. Dacă notăm cu  $p$  semiperimetrul triunghiului, atunci aria se calculează cu ajutorul expresiei:

$$\text{aria} = \sqrt{p(p-a)(p-b)(p-c)}$$

Amintim că măsurile laturilor triunghiului satisfac relațiile

$$p-a > 0, p-b > 0, \text{ și } p-c > 0$$

deci se va testa condiția

$$p-a > 0 \ \&\& \ p-b > 0 \ \&\& \ p-c > 0$$

care trebuie să fie adevărată.

## PROGRAMUL BIV23

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main() /* citește măsurile laturilor unui triunghi și afișează aria lui */
{
    double a,b,c,p;
    char tab[255];

    do { /* citește măsurile laturilor triunghiului */
        do { /* citește pe a */
            printf("a= ");
            if(gets(tab) == NULL) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(tab,"%lf",&a) == 1 && a>0)
                break; /* se iese din ciclul pentru citirea lui a */
            printf("nu s-a tastat un numar pozitiv\n");
            printf("se reia citirea lui a\n");
        } while (1);
        do { /* citește pe b */
            printf("b= ");
            if(gets(tab) == NULL) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
        }
        if(sscanf(tab,"%lf",&b) == 1 && b>0)
```

```

        break; /* se iese din ciclul do-while deoarece s-a
               citit valoarea lui b */
    printf("nu s-a tastat un numar pozitiv\n");
    printf("se reia citirea lui b\n");
} while (1);
do {
    printf("c= ");
    if(gets(tab) == NULL ) {
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if(sscanf(tab,"%lf",&c) == 1 && c>0)
        break; /* se iese din ciclul do-while deoarece
               s-a citit valoarea lui c */
    printf("nu s-a tastat un numar pozitiv\n");
    printf("se reia citirea lui c\n");
} while(1);
p = (a + b + c) / 2;
if( p-a > 0 && p -b > 0 && p - c >0 )
    break; /* se iese din ciclul do-while exterior deoarece a,b,c
           pot fi masurile laturilor unui triunghi */
printf("a= %g\tb =%g\tc= %g\n",a,b,c);
printf("nu pot reprezenta masurile laturilor\
unui triunghi\n");
} while (1);
printf("aria = %g\n",sqrt(p*(p-a)*(p-b)*(p-c)) );
}

```

4 24 Să se scrie un program care citește două numere naturale și pozitive, calculează și afișează cel mai mare divizor comun al lor (c m.m.d.c.).

Calculul celui mai mare divizor comun a două numere se poate face folosind algoritmul lui Euclid. Pașii acestui algoritm sînt:

1. Se citesc cele două numere.  
Fie acestea  $a$  și  $b$ .
2. Se împarte  $a$  la  $b$ ; fie  $q$  și  $r$  citul, respectiv restul acestei împărțiri:  
 $a=q*b+r$ .
3. Dacă  $r$  este diferit de zero, se fac atribuiri:  
 $a=b$  și  $b=r$   
și se revine la pasul 2.

Altfel algoritmul se intrerupe și cel mai mare divizor comun al lor este  $b$ .

## PROGRAMUL BIV24

```
#include <stdio.h>
#include <stdlib.h>

main() /* citește două numere întregi și pozitive
        și afișează c.m.m.d.c al lor */
{
    long m,n;
    long a,b;
    long r;
    char tab[255];

    do {
        printf("primul numar= ");
        if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(tab,"%ld",&m) == 1 && m > 0 )
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
    } while(1);

    do {
        printf("al doilea numar= ");
        if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(tab,"%ld",&n) == 1 && n > 0 )
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
    } while(1);

    a = m;
    b = n;

    do {
        r = a % b;
```

```

        if(r) {
            a = b;
            b = r;
        }
    } while(r);

    printf("(%ld,%ld) = %ld\n", m,n,b);
}

```

## 4.12. Instrucțiunea switch

Instrucțiunea *switch* permite realizarea structurii *selective*. Aceasta este o generalizare a structurii *alternative* și a fost introdusă de C.A.R. Hoare. Ea poate fi realizată prin instrucțiuni *if* imbricate. Utilizarea instrucțiunii *switch* face ca programul să fie mai clar decât dacă se utilizează varianta cu instrucțiuni *if* imbricate.

Structura *selectivă*, în forma în care a fost ea acceptată de către adepții programării structurate, se realizează în limbajul C cu ajutorul următorului format al instrucțiunii *switch*:

```

switch(expresie) {
    case c1:
        sir_1
        break;
    case c2:
        sir_2
        break;
    ..
    case cn:
        sir_n
        break;
    default:
        sir
}

```

unde.

*c1, c2, ..., cn* - Sînt constante.

*sir\_1, sir\_2, ..., sir\_n* - Sînt succesiuni de instrucțiuni.

Instrucțiunea *switch* cu formatul indicat mai sus se execută astfel:

1. Se evaluează expresia din parantezele rotunde.
2. Se compară pe rând valoarea expresiei cu valorile constantelor *c1*, *c2*, ..., *cn*.
  - Dacă valoarea expresiei coincide cu una din constante să zicem cu *c1*, atunci se execută secvența *sir\_1*, apoi se trece la instrucțiunea următoare instrucțiunii *switch*, adică la instrucțiunea aflată după acolada închisă care termină instrucțiunea.
  - Dacă valoarea respectivă nu coincide cu nici una din constantele *c1*, *c2*, ..., *cn*, atunci se execută succesiunea de instrucțiuni *sir* și apoi se ajunge la instrucțiunea următoare instrucțiunii *switch*.

Menționăm că este posibil să nu se ajungă la instrucțiunea următoare instrucțiunii *switch* în cazul în care succesiunea de instrucțiuni selectată pentru execuție (*sir\_i* sau *sir*) va defini ea însăși un alt mod de continuare a execuției programului (de exemplu, execuția instrucțiunii de revenire dintr-o funcție, saltul la o instrucțiune etichetată etc.)

Succesiunile *sir*, *sir\_1*, ..., *sir\_n* se numesc *alternativele* instrucțiunii *switch*.

Alternativa *sir* este opțională, deci într-o instrucțiune *switch*, secvența

**default:**  
*sir*

poate fi absentă.

În acest caz, dacă valoarea expresiei nu coincide cu valoarea nici uneia dintre constantele *c1*, *c2*, ..., *cn*, atunci instrucțiunea *switch* nu are nici un efect și se trece la execuția instrucțiunii următoare.

Instrucțiunea *switch* de mai sus este echivalentă cu următoarea instrucțiune *if* imbricată:

```
if (expresie == c1)
    sir_1
else
    if (expresie == c2 )
        sir_2
    else
        if (expresie == c3 )
            sir_3
        else
            if
                ...
```

```

else
    if( expresie == cn )
        sir_n
    else
        sir

```

Instrucțiunea *break* de la sfârșitul fiecărei alternative, după secvențele *sir\_1*, *sir\_2*, ..., *sir\_n*, permite ca la întâlnirea ei să se treacă la execuția instrucțiunii următoare instrucțiunii *switch*. Se obișnuiește să se spună că instrucțiunea *break* permite ieșirea din instrucțiunea *switch*.

Amintim că instrucțiunea *break* poate fi utilizată numai în corpurile ciclurilor și în alternativele instrucțiunii *switch*.

Prezența ei la sfârșitul fiecărei alternative a unei instrucțiuni *switch*, nu este obligatorie. În cazul în care instrucțiunea *break* este absentă la sfârșitul unei alternative, după execuția succesiunii de instrucțiuni din compunerea alternativei respective se va trece la execuția succesiunii de instrucțiuni din alternativa următoare a aceleiași instrucțiuni *switch*.

Astfel, dacă o instrucțiune *switch* are formatul:

```

switch(expresie){
    case c1 :
        sir_1
    case c2 :
        sir_2
}

```

atunci ea este echivalentă cu următoarea secvență.

```

if(expresie == c1){
    sir_1
    sir_2
}else
    if( expresie == c2 ) {
        sir_2
    }

```

#### Exerciții:

- 4.25 Să se scrie un program care citește o cifră din intervalul [1,7] și afișează denumirea zilei din săptămână corespunzătoare cifrei respective (1-luni, 2-marți etc).

## PROGRAMUL BIV25

```
#include <stdio.h>
#include <stdlib.h>

main() /* citește o cifră din intervalul [1,7] și afișează denumirea
       zilei din săptămâna corespunzătoare cifrei respective */
{
    int i;
    char t[255];

    do {
        puts("tastati o cifra din intervalul [1,7]");
        if(gets(t) == NULL ){
            puts("s-a tastat EOF");
            exit(1);
        }
        if(sscanf(t,"%d", &i) == 1 && i > 0 && i < 8 )
            break;
        puts("nu s-a tastat o cifra din\
            intervalul  [1,7]");
    } while(1);
    switch(i) {
        case 1:
            puts("luni");
            break;
        case 2:
            puts("marti");
            break;
        case 3:
            puts("miercuri");
            break;
        case 4:
            puts("joi");
            break;
        case 5:
            puts("vineri");
            break;
        case 6:
            puts("sambata");
            break;
        case 7:
            puts("duminica");
    }
}
```



}

4.26 Să se scrie un program care citește construcții de forma:

op1 op op2

unde:

op1 și op2 - Sînt întregi de tip *long*.

op - Este unul din caracterele: +, -, \* sau /.

și afișează valoarea expresiei citite.

Operația de împărțire este împărțirea întreagă.

Acest exemplu este întâlnit frecvent în diferite lucrări relativ la limbajul C. De exemplu, autorii limbajului C, în lucrarea [2] propun acest exemplu pentru a simula un calculator rudimentar de birou.

## PROGRAMUL BIV26

```
#include <stdio.h>
#include <stdlib.h>
```

```
main() /* citește o expresie de forma: op1opop2
      unde op1 și op2 sînt întregi de tip long, iar op este unul din
      operatorii +, -, *, /
      afișează valoarea expresiei respective */
{
    long op1, op2, rez;
    char op;
    char t[255];

    for ( ; ; ) {
        do {
            printf("tastati expresia:op1opop2: ");
            if( gets(t) == NULL )
                exit(0);
            if(sscanf(t, "%ld %c %ld", &op1, &op, &op2)
                == 3)
                break;
            printf("expresie eronata\n");
        } while(1);

        switch(op) {
            case '+' :
```

```

        rez=op1 + op2;
        break;
    case '-' :
        rez=op1 - op2;
        break;
    case '*' :
        rez= op1 * op2;
        break;
    case '/' :
        if(op2 == 0 ) {
            printf("divizor nul\n");
            rez = 0;
        }
        else
            rez = op1/op2;
        break;
    default:
        printf("operator eronat\n");
        rez = 0;
} /* sfirsit switch*/

printf("op1= %ld op= %c op2= %ld rez=%ld\n",
        op1,op,op2,rez);
} /* sfirsit for */
} /* sfirsit main */

```

#### Observație:

Programul de față permite să se evalueze mai multe expresii separate prin caractere albe. El își întrerupe execuția la întâlnirea sfârșitului de fișier.

## 4.13. Instrucțiunea goto

Instrucțiunea *goto* nu este o instrucțiune absolut necesară la scrierea programelor în limbajul C. Cu toate acestea, ea se dovedește utilă în anumite cazuri. Autorii limbajului recomandă utilizarea ei în cazul în care se dorește să se iasă din mai multe cicluri imbricate. Astfel de situații apar adesea la întâlnirea unei erori. În astfel de situații, de obicei, se dorește să se facă un salt în afara ciclurilor în care a intervenit eroarea, pentru a se ajunge la o secvență externă lor de tratare a erorii respective.

Înainte de a indica formatul instrucțiunii *goto* să precizăm noțiunea de *etichetă*.

Prin *etichetă* înțelegem un nume urmat de două puncte:

nume:

*nume* utilizat în definirea unei etichete este numele etichetei respective.

După o etichetă urmează o instrucțiune.

Se obișnuiește să se spună că eticheta *prefixează* instrucțiunea care urmează după ea.

Etichetele sînt *locale* în corpul funcției în care sînt definite.

Instrucțiunea *goto* are formatul:

*goto* *nume*;

unde:

*nume* - Este numele unei etichete definită în corpul aceleiași funcții în care se află instrucțiunea *goto*.

La întâlnirea instrucțiunii *goto*, se realizează un salt la instrucțiunea prefixată de eticheta al cărei nume se află după cuvîntul cheie *goto*.

Deoarece o etichetă este locală în corpul unei funcții rezultă că ea este nedefinită în afara corpului funcției respective. În felul acesta, o instrucțiune *goto* poate realiza un salt numai la o instrucțiune din corpul aceleiași funcții în care este ea utilizată.

Deci, o instrucțiune *goto* nu poate face salt din corpul unei funcții la o instrucțiune din corpul altei funcții.

Menționăm că nu se justifică utilizarea abuzivă a acestei instrucțiuni. Se recomandă a fi utilizată pentru a simplifica ieșirea din cicluri imbricate.

**Exemplu:**

Presupunem că într-un punct al programului, aflat în interiorul mai multor cicluri, se depistează o eroare și se dorește să se continue execuția programului cu o secvență de tratare a erorii respective. În acest caz, vom folosi o instrucțiune *goto* ca mai jos.

```
for (...) {  
    ...  
    while (...) {  
        ...  
        do {  
            ...  
            for (...) {  
                ...  
            }  
        }  
    }  
}
```

```

        if(i==0)
            goto divzero;
        else
            x=y/i;
            ...
    }
    ...
}while(...);
...
}
...
}
...

/* secventa de tratare a erorii */
divzero:
    printf(...);
    ...

```

În absența instrucțiunii *goto* se poate realiza același lucru folosind un indicator și o serie de teste realizate asupra lui.

## 4.14. Programarea procedurală, funcții, apelul și revenirea din ele

Începînd cu primele limbaje de programare de nivel înalt s-a utilizat *programarea procedurală*.

Aceasta s-a dezvoltat din necesitatea de a utiliza într-un program, aceeași secvență de calcul de mai multe ori. Pentru a evita o astfel de repetiție, secvența de instrucțiuni corespunzătoare se organizează ca o parte distinctă și se face un salt la ea, ori de câte ori este nevoie în program de procesul de calcul respectiv. Acest salt este cu revenire la instrucțiunea următoare instrucțiunii care a făcut saltul și de aceea el diferă de salturile realizate prin instrucțiunea *goto*. Secvența de instrucțiuni organizată în acest fel are diferite denumiri în diverse limbaje de programare: sub-program, subrutină, procedură etc. Pentru început considerăm denumirea de *procedură*.

Uneori procedura trebuie să exprime același proces de calcul dar care se realizează cu date diferite. În acest caz, procedura trebuie realizată generală, făcînd *abstracție* de datele respective. De exemplu, pentru a evalua expresia:

(1)  $4^{**10} - 3^{**20}$

putem construi o procedură pentru ridicarea la putere, care să fie generală și să facă abstracție de valorile efective pentru bază și exponent: 4 și 10 pentru prima ridicare la putere, 3 și 20 pentru cea de a doua. Această generalizare se realizează considerind ca fiind variabile atât baza cât și exponentul, iar valorile lor se precizează la fiecare apel al procedurii implementate în acest fel.

Aceste variabile utilizate pentru a putea implementa o procedură generală și care se concretizează la fiecare apel al procedurii, se numesc *parametri formali*.

În felul acesta, procedura apare ca un rezultat al unui proces de generalizare necesar implementării ei. Ca orice proces de generalizare ea presupune o abstractizare care se realizează prin utilizarea parametrilor formali.

Valorile de la apel ale parametrilor formali se numesc *parametri efectivi*.

Programarea *procedurală* are la bază utilizarea procedurilor, iar acestea la rândul lor realizează o abstractizare prin parametri.

În lucrarea [14] se subliniază și un alt aspect al abstractizării realizate prin utilizarea procedurilor și anume acela că o procedură se poate asemana cu o *cutie neagră*, la care i se transferă date la apel, iar aceasta furnizează rezultatele la revenirea din ea. În momentul apelului, se face abstracție de metoda de prelucrare a datelor de intrare în rezultate care se mai numesc și *date de ieșire*.

În toate limbajele de programare se consideră două categorii de proceduri:

- a. Proceduri care definesc o valoare de revenire
- b. Proceduri care nu definesc o valoare de revenire.

Procedurile din categoria a. de obicei se numesc *funcții*. *Valoarea de revenire* se mai numește și *valoare de întoarcere* sau *valoarea returnată* de funcție.

Procedura pentru calculul ridicării la putere este un exemplu de funcție. Ea are ca parametrii bază și exponentul, iar ca valoare de întoarcere sau returnată, rezultatul ridicării valorii bazei la valoarea exponentului, valori care sînt definite la apel.

În limbajele C și C++ atât procedurile din categoria a, cât și cele din categoria b. se numesc *funcții*. Deci, în aceste limbaje distingem funcții care returnează o valoare la revenirea din ele, precum și funcții care nu returnează nici o valoare.

O funcție are o *definiție* și atâtea *apeluri* într-un program, câte sînt necesare.

O definiție de funcție are formatul:

antet  
corp

unde:

*antet* - Are formatul:

*tip nume (lista declarațiilor parametrilor formali)*

*corp* - Este o instrucțiune compusă.

Amintim că *tip* este cuvîntul cheie *void* pentru funcții care nu returnează nici o valoare la revenirea din ele. Pentru alte detalii vezi paragraful 1.4.

Apelul unei funcții trebuie să fie precedat de definiția sau de prototipul ei.

Prototipul unei funcții conține informații asemănătoare cu cele din antetul ei:

- tipul valorii returnate;
- numele funcției;
- tipurile parametrilor.

El poate avea același format ca și antetul funcției, în plus este urmat de punct și virgulă. Pentru alte detalii vezi paragraful 1.11.

Limbajul C se livrează cu o serie de funcții care au o utilizare frecventă în programe. Ele se păstrează într-un fișier special în format OBJ, adică compilat și se adaugă la fiecare program în faza de editare. Aceste funcții sînt numite *funcții standard de bibliotecă*. Ele au prototipurile în diferite *fișiere de extensie .h*. Exemple de astfel de funcții sînt funcțiile cu ajutorul cărora se realizează operații de intrare/ieșire (printf, scanf, puts, gets, getch, putchar etc.), funcțiile pentru calculul funcțiilor elementare (sqrt, sin, cos, atan, log, pow etc.), funcții de conversii (sscanf, sprintf etc.) etc.

Prototipurile funcțiilor standard se includ în program înaintea apelurilor lor folosind construcția *#include* tratată prin preprocesor (vezi paragraful 1.12.).

**Exemple de fișiere cu prototipuri:**

stdio.h	pentru funcțiile printf, scanf, gets, puts etc.,
conio.h	pentru funcțiile putchar, getch, getche etc.;
math.h	pentru funcțiile elementare sqrt, sin, cos etc.

Menționăm că pînă în prezent în toate exercițiile au fost apelate numai funcții standard.

Apelul unei funcții care nu returnează o valoare, se realizează printr-o *instrucțiune de apel*. Aceasta are formatul:

*nume (lista parametrilor efectivi);*

unde,

*nume* - Este numele funcției care se apelează.

*lista parametrilor efectivi* - Este fie vidă, cînd funcția nu are parametri, fie o *expresie* sau mai multe separate prin *virgulă*.

Deci un *parametru efectiv* este o expresie.

Parametrii efectivi de la apel se corespund cu cei formali prin ordine. Primul parametru efectiv corespunde primului parametru formal, cel de al doilea parametru efectiv corespunde celui de al doilea parametru formal ș.a.m.d.

Amintim că parametrii unei funcții (formali sau efectivi) se mai numesc și *argumente*.

O funcție care returnează o valoare, poate fi apelată fie printr-o *instrucțiune de apel*, fie ca *operand* al unei expresii. În cazul în care funcția se apelează printr-o *instrucțiune de apel*, se pierde valoarea returnată. Cînd funcția se apelează ca *operand* al unei expresii, valoarea returnată de ea se utilizează la evaluarea expresiei respective

Spre exemplificare să considerăm funcția *getch*. Noi am apelat această funcție atît ca *operand* în expresii de atribuire de forma:

```
c=getch()
```

cît și printr-o *instrucțiune de apel* de forma:

```
getch();
```

În primul caz, valoarea codului ASCII al caracterului citit de la tastatură se atribuie variabilei *c*. La cel de al doilea apel, valoare codului ASCII al caracterului citit de la tastatură nu este utilizată. În acest caz apelul se face cu scopul de a afișa ecranul utilizator și de a bloca execuția programului pînă la acționarea unei taste oarecare, corespunzătoare caracterelor ASCII.

La apelul unei funcții, valorile parametrilor efectivi se atribuie parametrilor formali corespunzători, apoi execuția continuă cu prima *instrucțiune* din corpul funcției apelate.

În cazul în care tipul unui parametru efectiv diferă de tipul parametrului

formal care-i corespunde, în limbajul C se convertește automat valoarea parametrului efectiv spre tipul parametrului formal respectiv. În limbajul C++ se utilizează o regulă mai complexă pentru apelul funcțiilor și de aceea se recomandă să nu se folosească nici în limbajul C, regula de conversie automată amintită mai sus. În acest scop se poate utiliza operatorul de conversie explicită ((tip)), adică folosind așa numitele expresii *cast*.

De exemplu, dacă funcția *f* are un parametru de tip *double* și *n* este o variabilă de tip *int*, atunci în locul apelului:

*f(n);*

se recomandă a folosi apelul cu conversie explicită a lui *n* spre tipul *double*:

*f((double)n);*

Acest al doilea apel se realizează identic în ambele limbaje

La revenirea dintr-o funcție apelată printr-o instrucțiune de apel se va continua cu execuția instrucțiunii următoare celei care a făcut apelul. În cazul în care o funcție este apelată ca un operand al unei expresii, la revenirea din ea se continuă cu evaluarea expresiei respective.

Revenirea dintr-o funcție se poate realiza în unul din următoarele două moduri:

- a. După execuția ultimei instrucțiuni din corpul funcției (s-a ajuns la acolada închisă care termină corpul funcției).
- b. La întâlnirea instrucțiunii *return*.

Instrucțiunea *return* este instrucțiunea de *revenire* dintr-o funcție. Ea are formatele:

1. ***return;***  
sau
2. ***return expresie;***

Cel de al doilea format se folosește în corpul unei funcții care returnează o valoare la revenirea din ea. Valoarea expresiei din instrucțiunea *return* este chiar valoarea returnată de funcție.

În cazul în care tipul acestei expresii diferă de tipul care precede numele din antetul funcției, valoarea expresiei se convertește automat spre tipul din antet, înainte de a se reveni din funcție.

Formatul 1 al instrucțiunii de revenire se utilizează numai în corpul unei funcții care nu returnează nici o valoare. Dintr-o astfel de funcție se poate



reveni și în modul indicat la punctul *a* de mai sus.

Faptul că instrucțiunea *return* (în ambele formate) poate fi scrisă în orice punct al corpului unei funcții, permite o mai mare flexibilitate în programarea în limbajele C și C++.

#### Exercițiul:

4.27 Să se scrie o funcție care are ca parametru un întreg *n* din intervalul [0,170], calculează și returnează pe *n!*

Numim *factorial* această funcție. Ea returnează o valoare flotantă în dublă precizie. Rezultă că funcția are antetul:

```
double factorial(int n)
```

La început funcția verifică dacă *n* aparține intervalului [0,170]. În cazul în care *n* nu aparține acestui interval, funcția va returna valoarea -1. Metoda de calcul este aceeași cu cea utilizată în exercițiul 4.12.

#### FUNCȚIA BIV27

```
double factorial(int n)
/* calculeaza si returneaza pe n! pentru n in intervalul [0,170];
   altfel returneaza -1 */
{
    double f;
    int i;

    if( n < 0 || n > 170)
        return -1.0;
    for( i=2, f=1.0; i <= n; i++)
        f *= i;
    return f;
}
```

4.28 Să se scrie un program care calculează și listează pe *m!* pentru *m*=0, 1, 2, ..., 170.

Acest program apelează funcția *factorial* definită mai sus pentru a-l calcula pe *m!* pentru o valoare dată a lui *m*.

Definiția funcției *factorial* și funcția principală care o apelează pot fi editate în același fișier sursă. Cele două funcții pot fi editate în orice ordine. În cazul în care definiția funcției *factorial* se află în fișierul sursă după funcția principală, apelul funcției *factorial* din funcția principală, trebuie să fie precedat de prototipul ei.

## PROGRAMUL BIV28

```
double factorial(int); /* prototipul functiei factorial */

#include <stdio.h>
#include <conio.h>

main() /* afiseaza pe m! pentru m = 0,1,2,...,170 */
{
    int m;

    for(m=0; m<171; m++) {
        printf("m=%d\tm!=%g\n",m,factorial(m));
        if((m+1)%23 == 0) {
            printf("actionati o tasta pentru a\
                continua\n");
            getch();
        } /* sfirsit if */
    } /* sfirsit for */
} /* sfirsit main */

double factorial(int n)
/* calculeaza si returneaza n! pentru n in intervalul [0,170];
   altfel returneaza -1 */
{
    double f;
    int i;

    if( n < 0 || n > 170)
        return -1.0;
    for( i=2, f=1.0; i <= n; i++)
        f *= i;
    return f;
} /* sfirsit factorial */
```

### Observații:

- 1 Funcția *factorial* este apelată ca parametru efectiv în apelul funcției *printf*. La apelul funcției *factorial*, valoarea parametrului ei efectiv *m* se atribuie parametrului formal *n*. Această valoare este apoi folosită în corpul funcției *factorial* pentru a-l calcula pe *n!* și deci și pe *m!*. Valoarea respectivă se obține ca valoare a lui *f*.

La revenirea din funcție se returnează valoarea lui *f*, adică chiar *m!*, care se afișează folosind specificatorul de format *%g*.

2. Cele două funcții pot fi editate în fișiere separate. De exemplu, presupunem că funcția *factorial* se editează în fișierul BIV27.CPP, iar funcția principală în fișierul BIV28A.CPP. În acest caz putem include fișierul BIV27.CPP folosind construcția *#include* pentru a compila împreună cele două funcții. Includerea se poate face înainte de definiția funcției *main* sau după ea. În cazul în care facem includerea în fața funcției *main*, nu mai este necesar să indicăm prototipul funcției *factorial* deoarece apelul ei este precedat chiar de definiția funcției. Procedând în acest fel se obține varianta de mai jos.

## PROGRAMUL BIV28A

```
#include <stdio.h>
#include <conio.h>
#include "biv27.cpp"

main() /* afiseaza pe m! pentru m = 0,1,2,...,170 */
{
    int m;

    for(m=0; m<171; m++) {
        printf("m=%d\tm!=%g\n",m,factorial(m));
        if((m+1)%23 == 0) {
            printf("actionati o tasta pentru a\
                continua\n");
            getch();
        }
    }
}
```

3. O altă posibilitate de compilare și link-editare a funcțiilor unui program, editate în mai multe fișiere, este aceea de a utiliza un fișier de tip *Project* (cu extensia *.prj*). Un astfel de fișier, conține numele fiecărui fișier (împreună cu extensia lui) care se compilează și link-editează în vederea obținerii fișierului executabil al programului. În acest fișier pot fi indicate nu numai fișiere de tip sursă (cu extensia *.CPP*), ci și fișiere de tip obiect (cu extensia *OBJ*) sau chiar fișiere cu biblioteci de funcții, altele decât cele ale sistemului.

Fișierele de tip *project* pentru limbajele Turbo C și C++ nu sînt compatibile. Ele se construiesc folosind meniul *Project* al celor două sisteme

întregate de dezvoltare.

Avantajele fişierelor de tip *Project* constau în aceea că, la fiecare lansare se compilează în mod automat numai sursele în care s-au făcut modificări. De aceea, în cazul programelor sursă mari se recomandă împărţirea lor în mai multe fişiere sursă şi compilarea lor prin utilizarea fişierelor de tip *Project*. De obicei, într-un fişier sursă se grupează funcţii "înrudite", adică funcţii care prelucrează în comun subseturi de date sau sînt logic legate între ele.

Întrucît exerciţiile pe care le prezentăm nu sînt de dimensiuni mari, vom utiliza frecvent includerile de fişiere folosind construcţia *#include* a preprocesorului.

4.29 Să se scrie o funcţie care are ca parametri doi întregi  $x$  şi  $y$ , calculează şi returnează numărul aranjamentelor de  $x$  obiecte luate câte  $y$ .

La început funcţia testează dacă  $x$  aparţine intervalului  $[1,170]$ , iar  $y$  intervalului  $[1,x]$ . În caz de eroare, funcţia returnează valoarea -1.

Dacă notăm cu  $A(x,y)$  numărul aranjamentelor de  $x$  obiecte luate câte  $y$ , atunci:

$$A(x,y) = x \cdot (x-1) \cdot (x-2) \cdot \dots \cdot (x-y+1)$$

## FUNCȚIA BIV29

```
double aranjamente ( int x, int y)
/* calculeaza si returneaza numarul aranjamentelor de x obiecte luate
   cite y */
{
    double a;
    int i;

    if(x < 1 || x > 170 )
        return -1.0;
    if(y < 1 || y > x )
        return -1.0;
    a = 1.0;
    i = x - y + 1;
    while( i <= x )
        a *= i++;
    return a;
}
```

4.30 Să se scrie un program care calculează și afișează numărul aranjamentelor de  $n$  obiecte luate câte  $k$ , pentru  $n=1,2,\dots,170$  și  $k=1,2,\dots,n$ .

Programul de față apelează funcția *aranjamente* definită în exercițiul precedent.

### PROGRAMUL BIV30

```
#include <stdio.h>
#include <conio.h>
#include "biv29.cpp" /* contine functia aranjamente */

main() /* calculeaza si afiseaza numarul aranjamentelor de n obiecte
        luate cate k, pentru n in intervalul [1,170] si k in
        intervalul [1,n] */
{
    int k,n;

    for(n = 1; n <= 170; n++) {
        printf("actionati o tasta pentru a\
                continua\n");
        getch();
        printf("\nn= %d\n",n);
        for(k=1; k <= n; k++) {
            printf("k=%d\tA(n,k)=%g\n",k,
                    aranjamente(n,k));
            if(k%23 == 0 ) {
                printf("actionati o tasta pentru a\
                        continua\n");
                getch();
            } /* sfirsit if */
        } /* sfirsit for interior */
    } /* sfirsit for exterior */
} /* sfirsit main */
```

4.31 Să se scrie o funcție care are ca parametri doi întregi  $x$  și  $y$ , calculează și returnează numărul combinărilor de  $x$  elemente luate câte  $y$ .

Funcția testează dacă  $x$  aparține intervalului  $[1,170]$ , iar  $y$  intervalului  $[0,x]$ . În caz de eroare, funcția returnează valoarea -1.

Pentru calculul numărului combinărilor de  $x$  elemente luate câte  $y$  se determină raportul dintre numărul aranjamentelor de  $x$  elemente luate câte  $y$  și  $y!$ .

În acest scop, funcția de față apelează funcțiile *aranjamente* și *factorial* definite în exercițiile 4.29. și respectiv 4.27.

### FUNCȚIA BIV31

```
#include "biv27.cpp"
#include "biv29.cpp"

double combinari(int x, int y )
/* calculeaza si returneaza numarul combinarilor de x obiecte
   luate cite y */
{
    if( x < 1 || x > 170 )
        return -1.0;
    if( y < 0 || y > x )
        return -1.0;
    if( y == 0 || y == x )
        return 1.0;
    return aranjamente(x,y)/factorial(y);
}
```

#### Observații:

1. Un calcul mai rapid se realizează dacă se ține seama de relația

$$c(x,y)=c(x,x-y)$$

unde prin  $c(x,y)$  s-a notat numărul combinărilor de  $x$  obiecte luate cite  $y$ . Această relație este utilă să se aplice pentru

$$y > x/2.$$

În acest scop ultima instrucțiune se înlocuiește cu.

```
if(y > x/2)
    return aranjamente (x,x-y)/factorial(x-y);
else
    return aranjamente (x,y)/factorial(y);
```

2. O altă variantă pentru calculul numărului combinărilor este relația:

$$c(x,y)=(x/1)*((x-1)/2)*((x-2)/3)*..*((x-y+1)/y)$$

În acest caz se evită calculul valorilor  $A(x,y)$  și  $y!$  care cresc rapid odată cu creșterea valorilor lui  $x$  și  $y$ . Aceasta se realizează cu ajutorul funcției de mai jos.

## FUNCȚIA BIV31A

```
double combinari(int x, int y )
/* calculeaza si returneaza numarul combinarilor de x obiecte
   luate cite y */
{
    int i;
    double c;

    if( x < 1 || x > 170 )
        return -1.0;
    if( y < 0 || y > x )
        return -1.0;
    if( y == 0 || y == x )
        return 1.0;
    c = 1.0;
    for( i = 1; i <= y; i++)
        c = (c * (x - i + 1)) / i;
    return c;
}
```

- 4.32 Să se scrie un program care calculează și afișează numărul combinațiilor de  $n$  obiecte luate cite  $k$ , pentru  $n = 1, 2, \dots, 170$  și  $k = 0, 1, 2, \dots, n$ .

Programul utilizează funcția BIV31A.

## PROGRAMUL BIV32

```
#include <stdio.h>
#include <conio.h>
#include "biv31a.cpp"

main() /* afișeaza numarul combinarilor de n obiecte luate cite k
       pentru n in intervalul [1,170] si k in intervalul [0,n] */
{
    int k,n;

    for(n=1; n<= 170; n++) {
        printf("actionati o tasta pentru a\
              continua\n");
        getch();
        printf("\nn=%d\n",n);
        for( k=0; k <=n ; k++) {
            printf("k=%d\tC(n,k)=%g\n", k,
```

```

        combinari(n,k));
    if((k+1)%23 == 0) {
        printf("actionati o tasta pentru a\
        continua\n");
        getch();
    } /* sfirsit if */
} /* sfirsit for interior */
} /* sfirsit for exterior */
} /* sfirsit main */

```

- 4.33 Să se scrie o funcție care ridică la o putere întreagă nenegativă un număr flotant de tip *long double*.

Funcția utilizează metoda expusă în exercițiul 4.19.

### FUNCȚIA BIV33

```

long double ldrp( long double x, int n)
/* ridica pe x la puterea n */
{
    long double f,p;

    for(p=x,f=1.0L; n ; n >= 1) {
        if( n%2)
            f *= p;
        if( n > 1)
            p *= p;
    }
    return f;
}

```

- 4.34 Să se scrie un program care tabelizează puterile întregi ale unui număr flotant de tip *long double*, exponentul variind, cu pasul 1, între două limite  $m$  și  $n$ .

### PROGRAMUL BIV34

```

#include <stdio.h>
#include <stdlib.h>
#include "biv33.cpp"

main() /* tableaza puterile intregi ale unui numar*/
{

```



```

char t[255];
long double f,g;
int i,m,n;

do {
    printf("tastati numarul care se ridica la\
           puteri: ");
    if( gets(t) == NULL ) {
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if(sscanf(t,"%Lf",&f) == 1 )
        break;
    printf("nu s-a tastat un numar\n");
} while(1);

do {
    do {
        printf("tastati limita inferioara a\
               exponentului: ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d",&m) == 1)
            break;
        printf("nu s-a tastat un intreg\n");
    } while (1);

    do {
        printf("tastati limita superioara a\
               exponentului: ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d", &n ) == 1 )
            break;
        printf("nu s-a tastat un intreg\n");
    } while(1);

    if(m <= n)
        break;
    printf("limita inferioara:%d o depaseste\

```

```

        pe cea superioara:%d\n", m,n);
} while (1);
if( f == 0 && m < 0 ) {
    printf("putere negativa pentru zero\n");
    exit(1);
}
for( i=m; i <= n; i++) {
    if( i < 0 )
        g = 1.0/ldrp(f,-i);
    else
        g = ldrp(f,i);
    printf("%Lg la puterea %d este: %Lg\n",f,i,g);
}
}

```

## 4.15. Apel prin valoare și apel prin referință

În limbajul C, la apelul unei funcții, fiecărui parametru formal  $i$  se atribuie *valoarea* parametrului efectiv care-i corespunde. Deci, la apelul unei funcții se transferă valorile parametrilor efectivi. Din această cauză, se spune că apelul este prin *valoare* (*call by value*).

**Exemplu:**

Fie funcția *factorial* de prototip:

**double factorial(int n);**

definită în exercițiul 4.27. La apelul:

**fact=factorial(10);**

se atribuie parametrului formal  $n$  valoarea 10. Această atribuire se face înainte de execuția primei instrucțiuni a funcției *factorial*. De aceea, prin acest apel funcția factorial calculează pe 10!

În unele limbaje de programare, la apel nu se transferă valorile parametrilor efectivi, ci *adresele* acestor valori. În acest caz, se spune că apelul este prin *referință* (*call by reference*).

Între cele două tipuri de apeluri există o diferență esențială și anume:

- în cazul apelului prin valoare, funcția apelată nu poate modifica parametri efectivi din funcția care a făcut apelul, neavând acces la ei;
- în cazul apelului prin referință, funcția apelată, dispunând de adresa parametrilor efectivi, îi poate modifica.

Așa se întâmplă, de exemplu, în cazul limbajului FORTRAN, unde

apelul este prin referință.

De aici, rezultă că la apelul prin valoare transferul datelor este *unidirecțional*, adică valorile se transferă prin parametri numai de la funcția care face apelul spre cea care a fost apelată, nu și invers.

Există situații când se dorește ca funcția apelată să modifice valorile parametrilor efectivi. Unele limbaje permit ambele tipuri de apeluri. Așa este, de exemplu, cazul limbajului Pascal. Parametri declarați în Pascal prin *var* se transferă prin referință.

În limbajul C++, spre deosebire de limbajul C, există ambele apeluri, ca și în limbajul Pascal.

Apelul prin valoare este util în cazul în care funcția apelată nu trebuie să modifice valorile parametrilor efectivi, deoarece în felul acesta parametrii efectivi sînt protejați față de eventualele modificări care ar putea apare din greșeală.

Aceasta înseamnă că o funcție poate să modifice parametrii ei formali fără riscul ca prin modificarea respectivă să influențeze valorile parametrilor efectivi corespunzători. Așa de exemplu, funcția *ldrp*, definită în exercițiul 4.33, modifică valoarea parametrului formal *n* la zero. Cu toate acestea, valoarea parametrului efectiv corespunzător lui *n* nu suferă nici o modificare, avînd aceeași valoare la revenirea din funcție ca și în momentul apelului.

În limbajul C, un parametru efectiv poate fi numele unui tablou. De exemplu, fie o funcție, pe care o numim *sum* și care însumează elementele unui tablou de tip *int*. Această funcție are doi parametri:

- tab* - Este numele tabloului ale cărui elemente se însumează.
- n* - Este numărul elementelor tabloului.

Ea returnează suma:

$tab[0] + tab[1] + \dots + tab[n]$ .

Conform celor spuse mai sus, funcția are următorul antet:

`int sum(int tab[], int n)`

Amintim că dacă un parametru efectiv este un nume de tablou unidimensional, atunci parametrul formal corespunzător se poate declara fără a indica, în parantezele pătrate, limita superioară a indicelui său.

Dacă *a* este un tablou declarat ca mai jos:

`int a[10];`

atunci pentru a însuma elementele tabloului respectiv vom putea folosi

apelul:

```
x=sum(a,10);
```

Deoarece *a* este numele unui tablou, *a* are ca valoare adresa primului său element, adică adresa lui *a*[0]. Apelul fiind prin valoare, parametrul *tab* primește valoarea lui *a* deci *tab* are și el ca valoare adresa lui *a*[0]. Rezultă că *tab*[0], *tab*[1], ..., *tab*[9] reprezintă aceleași elemente ca *a*[0], *a*[1], ..., *a*[9]. De aceea, corpul funcției *sum* referindu-se la elementele lui *tab*, se referă de fapt chiar la elementele lui *a*:

```
for (s=0; n>0; n--)  
    s+=tab[n-1];
```

Se observă că, în acest caz, deși apelul s-a făcut prin valoare (valoarea lui *a* s-a atribuit lui *tab*), s-a realizat un apel prin referință, deoarece valoarea atribuită parametrului formal este o adresă.

În acest caz, funcția apelată poate modifica elementele tabloului al cărui nume s-a folosit ca parametru efectiv.

Într-adevăr, în exemplul de mai sus, atribuirea:

```
tab[3]=100;
```

scrisă în corpul funcției *sum* are ca efect atribuirea valorii 100 elementului *a*[3].

În concluzie, în limbajul C, apelul este prin valoare. Acest apel devine prin referință în cazul în care parametrul efectiv este numele unui tablou.

Ulterior o să vedem că apelul prin referință poate fi realizat în limbajul C și în cazul variabilelor simple, folosind *pointeri*.

De asemenea, într-un alt capitol o să vedem că în limbajul C++ putem să stabilim, pentru fiecare parametru, tipul de apel: prin valoare sau prin referință.

#### Exerciții:

4.35 Să se scrie o funcție care calculează valoarea unui polinom folosind schema lui Horner.

Parametrii funcției sînt:

- |          |  |
|----------|--|
| <i>c</i> | - Tabloul cu coeficienții polinomului. |
|          | - Este de tip <i>long double</i> .     |
| <i>n</i> | - Gradul polinomului.                  |
|          | - Este de tip <i>int</i> .             |
| <i>x</i> | - Valoarea variabilei.                 |

- Este de tip *long double*.

Tabloul  $c$  conține coeficienții polinomului în ordine descrescătoare a puterilor lui  $x$ .

- $c[0]$  - Este coeficientul lui  $x^{*n}$ .
- $c[1]$  - Este coeficientul lui  $x^{*(n-1)}$ .
- ...
- $c[n-1]$  - Este coeficientul lui  $x$ .
- $c[n]$  - Este termenul liber.

Se știe că un polinom  $p(x)$ , cu coeficienții  $c[0], c[1], \dots, c[n]$ , în ordine descrescătoare ale puterilor lui  $x$ , se poate pune sub forma:

$$p(x) = (( (c[0]*x + c[1])*x + c[2])*x + \dots + c[n-2])*x + c[n-1])*x + c[n]$$

Fie

$$s = c[0]$$

Se observă că paranteza cea mai interioară are ca valoare, valoarea expresiei:

$$s*x + c[1]$$

Să atribuim lui  $s$  valoarea acestei expresii.

$$s = s*x + c[1]$$

În continuare trebuie să se evalueze valoarea expresiei.

$$s*x + c[2]$$

Atribuim din nou lui  $s$  valoarea acestei expresii:

$$s = s*x + c[2]$$

În general, la al  $i$ -lea pas se evaluează expresia

$$s*x + c[i]$$

și apoi valoarea ei se atribuie lui  $s$ :

$$s = s*x + c[i]$$

Deci, calculul valorii polinomului  $p(x)$  revine la execuția repetată a atribuirii

$$s = s*x + c[i], \text{ pentru } i = 1, 2, \dots, n,$$

valoarea inițială a lui  $s$  fiind  $c[0]$ . Aceasta conduce la instrucțiunea *for* de mai jos:

```
for(s=c[0], i=1; i<=n; i++)
    s=s*x+c[i];
```

## FUNCȚIA BIV35

```
long double polinom(long double x, int n,
    long double c[])
/* calculeaza valoarea polinomului

$$p(x) = c[0]*x^n + c[1]*x^{(n-1)} + \dots + c[n-1]*x + c[n] */$$

{
    long double s;
    int i;

    for(s = c[0], i = 1; i <= n; i++)
        s = s*x + c[i];
    return s;
}
```

### Observație:

Biblioteca standard a limbajului C conține funcția *poly* care calculează valoarea unui polinom de gradul  $n$  cu coeficienții de tip *double*. Ea are prototipul:

```
double poly(double x, int n, double c[]);
```

care se află în fișierul *math.h*.

În acest caz,  $c[0]$  este termenul liber,  $c[1]$  coeficientul lui  $x$ . În general  $c[i]$  este coeficientul lui  $x$  la puterea  $i$ .

4.36 Să se scrie un program care realizează următoarele:

- citește pe  $x$  de tip *long double*;
- citește pe  $n$  de tip *int* din intervalul  $[0,100]$ ;
- citește coeficienții  $a[0]$ ,  $a[1]$ , ...,  $a[n]$  ai polinomului  $p(x)$  de tip *long double*;
- calculează și afișează valoarea polinomului  $p(x)$ .

$a[0]$  este coeficientul lui  $x^n$ ,  $a[1]$  este coeficientul lui  $x^{(n-1)}$  etc.

### PROGRAMUL BIV35

```
#include <stdio.h>
#include <stdlib.h>
```

```

#include "biv35.cpp"

#define MAXGRAD 100
main() /* citește pe x, n, a[0], a[1], ..., a[n], calculează și
        afișează valoarea polinomului
        
$$p(x) = a[0] \cdot x^n + a[1] \cdot x^{n-1} + \dots + a[n-1] \cdot x + a[n]$$
 */
{
    long double x, a[MAXGRAD+1];
    int i, n;
    char t[255];

    /* citește valoarea lui x */
    do {
        printf("valoarea lui x: ");
        if( gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t, "%Lf", &x) == 1 )
            break;
        printf("nu s-a tastat un număr\n");
    } while(1);

    /* citește pe n */
    do {
        printf("valoarea lui n: ");
        if( gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t, "%d", &n) == 1 && n >= 0 &&
            n <= 100 )
            break;
        printf("nu s-a tastat un întreg din\
            intervalul [0,100]\n");
    } while (1);

    /* citește coeficienții polinomului */
    for( i=0; i <= n; i++ )
        do {
            printf("coeficientul a[%d]: ", i);
            if( gets(t) == NULL ) {
                printf("s-a tastat EOF\n");
            }
        }
    }

```

```

        exit(1);
    }
    if(sscanf(t,"%Lf", &a[i]) == 1)
        break; /* iesire din do-while */
    printf("nu s-a tastat un numar\n");
} while(1);
/* sfirsit citire coeficienti polinom */

printf("x=%Lg\tgrad=%d\tval pol=%Lg\n",x,n,
        polinom(x,n,a));
}

```

- 4.37 Să se scrie o funcție care citește cel mult  $n$  numere și le păstrează în tabloul *ndtab* care este de tip *double*. Funcția returnează numărul numerelor citite.

### FUNCȚIA BIV37

```

int ndcit(int n, double ndtab[] )
/* citește cel mult n numere și le păstrează în tabloul ndtab;
   returnează numărul numerelor citite */
{
    int i;
    double d;
    char t[255];

    for(i=0; i < n; i++) {
        printf("elementul [%d]= ",i);
        if(gets(t) == NULL )
            return i;
        if(sscanf(t,"%lf", &d) != 1)
            break;
        ndtab[i] = d;
    }
    return i;
}

```

- 4.38 Să se scrie o funcție care citește componentele unui vector precedate de numărul lor. Funcția returnează numărul componentelor vectorului respectiv.

Această funcție folosește funcția definită în exercițiul precedent.



## FUNCȚIA BIV38

```
int dvcit(int nmax, double dvector[])
/* citește un întreg n și cele n componente ale vectorului
   dvector; returnează numărul componentelor citite */
{
    int i,n;
    char t[255];

    do {
        printf("numarul elementelor= ");
        if(gets(t) == NULL) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d", &n) == 1 && n > 0 &&
           n <= nmax )
            break;
        printf("nu s-a tastat un intreg din\
              intervalul [1,%d]\n", nmax);
    } while(1);
    if((i=ndcit(n,dvector)) != n) {
        printf("nu s-a reusit citirea celor n=%d\
              componente\n",n);
        printf("s-au citit numai %d componente\n",i);
    }
    return i;
}
```

4.39 Să se scrie un program care citește componentele vectorilor  $x$  și  $y$ , calculează și afișează produsul lor scalar.

Componentele celor doi vectori se tastează astfel:

- numărul componentelor vectorilor ( $n$ );
- cele  $n$  componente ale lui  $x$ ;
- cele  $n$  componente ale lui  $y$ .

Programul de față utilizează funcția definită în exercițiul precedent.

## PROGRAMUL BIV39

```
#include <stdio.h>
#include <stdlib.h>
#include "biv37.cpp"
```

```

#include "biv38.cpp"

#define MAX 1000

main() /* citește componentele vectorilor x și y precedate de
        numărul lor; calculează și afișează produsul lor scalar */
{
    int i,n;
    double x[MAX],y[MAX],s;

    /* citește pe n și componentele vectorului x */
    n = dvcit(MAX,x);

    /* citește componentele vectorului y */
    if(ndcit(n,y) != n) {
        printf("nu sînt n=%d componente pentru\
                y\n",n);
        exit(1);
    }

    /* calculează produsul scalar */
    s = 0.0;
    for(i=0; i < n; i++)
        s += x[i]*y[i];
    printf("(x,y)= %g\n", s);
}

```

4.40) Să se scrie o funcție care sortează în ordine crescătoare, elementele unui tablou de tip *double*.

Spunem despre un tablou *tab*, de elemente numerice, că este sortat crescător (descrescător), dacă elementele lui satisfac relația:

$$\text{tab}[i] \leq (>=) \text{tab}[i+1]$$

pentru

$$i=0,1,2,\dots,n-2$$

unde:

*n* - Este numărul elementelor tabloului *tab*.

Funcția care realizează sortarea are doi parametri:

*tab* - Tabloul de sortat.

*n* - Numărul elementelor tabloului.

La ora actuală, există o serie de metode de sortare, care diferă între ele, mai ales, prin ordinul de mărime al pașilor necesari pentru a realiza sortarea. Pentru amănunte se poate consulta [3].

Cele mai simple metode sînt cele mai puțin eficiente și necesită un număr de pași de ordinul lui  $n^2$ ,  $n$  fiind numărul elementelor de sortat. Prezentăm mai jos o astfel de metodă, care este cunoscută sub denumirea de *metoda bulelor (bubble sort)*.

Ea constă în următoarele:

Se compară primele două elemente ale tabloului de sortat și dacă nu sînt în ordinea cerută (de exemplu crescătoare), se permută între ele. Apoi se compară elementul al doilea cu al treilea și se procedează ca mai sus. În general, se compară două elemente învecinate, să zicem al  $i$ -lea cu al  $i+1$ -lea și dacă nu sînt în ordinea cerută, atunci ele se *permută*. Se procedează în acest fel cu toate perechile de elemente învecinate ale tabloului de sortat. Apoi procesul respectiv se reia de la început. El se întrerupe cînd se constată că toate elementele învecinate ale șirului sînt în ordinea cerută (în cazul de față, crescătoare).

**Exemplu:**

Fie șirul de numere:

8,3,9,5,4,7

pe care dorim să-l ordonăm crescător.

*Prima parcurgere a șirului:*

8 3 9 5 4 7 șirul inițial

3 8 9 5 4 7 șirul după permutarea lui 8 cu 3

3 8 5 9 4 7 șirul după permutarea lui 9 cu 5

3 8 5 4 9 7 șirul după permutarea lui 9 cu 4

3 8 5 4 7 9 șirul după permutarea lui 9 cu 7.

*A doua parcurgere a șirului:*

3 8 5 4 7 9 șirul după prima parcurgere

3 5 8 4 7 9 șirul după permutarea lui 8 cu 5

3 5 4 8 7 9 șirul după permutarea lui 8 cu 4

3 5 4 7 8 9 șirul după permutarea lui 8 cu 7.

*A treia parcurgere a șirului:*

3 5 4 7 8 9 șirul după a doua parcurgere a lui

3 4 5 7 8 9 șirul după permutarea lui 5 cu 4.

La a patra parcurgere a șirului nu se mai face nici o permutare deoarece

perechile de elemente vecine sînt în ordine crescătoare, deci şirul este sortat crescător.

O proprietate a acestei metode este aceea că la fiecare parcurgere a şirului de numere, cel puţin un element ajunge să-şi ocupe poziţia sa finală în conformitate cu ordonarea avută în vedere. Astfel, după prima parcurgere a şirului, elementul ajuns pe ultimul loc nu va mai fi permutat, ocupîndu-şi locul final. În exemplul de mai sus, 9 a ajuns pe ultimul loc după prima permutare, poziţie care este finală pentru el. După cea de a doua parcurgere, 8 a ajuns şi el pe poziţia lui finală. Întîmplător şi 7 a ajuns pe poziţia lui finală la cea de a doua parcurgere şi aşa mai departe.

Dacă se ordonează crescător un tablou de numere, atunci după prima parcurgere a elementelor lui, elementul maxim va avea indicele cel mai mare. După cea de-a doua parcurgere, maximul dintre elementele rămase (fără a considera ultimul element) va ocupa penultima poziţie în tablou etc.

Se obişnuieşte să se spună că elementele urcă în poziţiile lor la fel ca şi bulele de aer în apa care fierbe, de unde şi denumirea metodei.

Funcţia de mai jos conţine un ciclu pentru parcurgerea elementelor tabloului *tab*. În acest ciclu se compară elementele vecine curente şi dacă acestea nu sînt în ordine crescătoare, ele se permută. Deoarece, după o parcurgere a şirului se reîncepe parcurgerea lui, acest ciclu este conţinut într-un altul care realizează execuţia repetată a ciclului de parcurgere a şirului. Se ajunge în acest fel la următorii paşi:

1. Cît timp şirul nu este sortat se execută pasul 2.

Altfel se întrerupe procesul de sortare.

2. Se parcurge şirul şi se permută perechile de elemente învecinate care sînt în ordine descrescătoare.

Pasul 2 se realizează printr-un ciclu *for* de forma:

```
for(i=0; i<n-1; i++)  
    if(tab[i]>tab[i+1]){  
  
        /* se permută elementele tab[i] şi tab[i+1] */  
        t=tab[i];  
        tab[i]=tab[i+1];  
        tab[i+1]=t;  
    }
```

Acest ciclu trebuie repetat atît timp cît elementele tabloului *tab* nu sînt în ordine crescătoare. Problema care se pune în legătură cu execuţia repetată a acestui ciclu *for* este aceea de a şti cînd s-a ajuns la situaţia în care

elementele tabloului *tab* sînt în ordine crescătoare. Observăm că în acest caz ciclul *for* se execută fără a mai face permutări. Deci, tabloul este sortat cînd ciclul *for* îl parcurge fără a face permutări. Acest fapt poate fi detectat folosind o variabilă, să zicem *ind*, care se anulează înaintea fiecărei execuții a ciclului *for* și se setează la valoarea 1 în cazul în care ciclul respectiv face o permutare. În felul acesta, variabila *ind* are valoarea 1 după execuția ciclului *for*, dacă acesta a făcut cel puțin o permutare și zero în caz contrar.

În concluzie, variabila *ind* are valoare zero cînd șirul este sortat și 1 în caz contrar. Variabila *ind* se gestionează astfel:

- inițial *ind* are valoarea 1;
- înainte de instrucțiunea *for*, *ind* se face egal cu zero;
- în secvența de permutare a două elemente învecinate, *ind* se face egal cu 1.

Se obține secvența:

```
ind = 0;
for (i=0; i<=n-1; i++)
    if (tab[i]>tab[i+1]){
        t=tab[i];
        tab[i]=tab[i+1];
        tab[i+1]=t;
        ind=1;
    }
```

Accastă secvență se execută repetat atît timp cît *ind* nu este egal cu zero, adică tabloul nu este sortat. În acest scop se folosește un ciclu *while* care are ca și corp chiar secvența de mai sus:

```
while(ind){
    ind=0;
    for(i=0; i<=n-1; i++)
        if (tab[i]>tab[i+1]){
            ...
        } /* sfîrsit if */
} /* sfîrsit while */
```

## FUNCȚIA BIV40

```
void ordcresc(double tab[], int n)
/* ordoneaza elementele lui tab in ordine crescatoare */
{
    int i, ind;
    double t;
```

```

ind = 1;
while(ind) {
    ind = 0;
    for(i=0; i< n-1; i++)
        if(tab[i] > tab[i+1]) {
            t = tab[i];
            tab[i] = tab[i+1];
            tab[i+1] = t;
            ind = 1;
        } /* sfirsit if */
    } /* sfirsit while */
}

```

- 4.41) Să se scrie un program care citește un șir de numere separate prin caractere albe și le afișează în ordine crescătoare. După ultimul număr se tastează un caracter nenumeric. Se presupune că la intrare există cel puțin două numere și cel mult 1000.

Numerele se citesc apelând funcția *ndcit* definită în exercițiul 4.37. Sortarea lor în ordine crescătoare se realizează folosind funcția *ordcresc* definită în exercițiul precedent.

## PROGRAMUL BIV41

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp"
#include "biv40.cpp"

#define MAX 1000

main() /* citește un șir de numere și le afișează în ordine crescătoare */
{
    double tnr[MAX];
    int n,i;

    if((n=ndcit(MAX,tnr)) < 2) {
        printf("s-au citit mai puțin de 2 numere\n");
        exit(1);
    }
    ordcresc(tnr,n);
    for(i=0; i < n; i++) {

```

```

printf("tnr[%d]=%g\n", i,tnr[i]);
if((i+1)%23 == 0) {
    printf("actionati o tasta pentru a\
        continua\n");
    getch();
}
}
)

```

4.42 Să se scrie o funcție care caută un element de valoare dată  $a$ , într-un tablou de numere sortat crescător. Funcția returnează indicele unui element de valoare egală cu  $a$  sau -1 în cazul în care nu există un astfel de element.

Elementul de valoare  $a$  poate fi găsit comparând fiecare element al tabloului cu  $a$  până când se găsește un element egal cu  $a$ . În cazul în care nu există un astfel de element în tabloul respectiv, compararea se face cu toate elementele tabloului pentru a putea constata că nu există elementul căutat. O astfel de metodă de căutare, numită căutare *liniară*, deși este foarte simplă, ea necesită un număr relativ mare de comparații.

În cazul tablourilor sortate se poate utiliza o metodă de căutare mult mai rapidă, numită metoda *căutării binare*. Această metodă constă în următoarele.

Fie  $tab$  tabloul în care se caută un element de valoare  $a$  și care este sortat crescător. Deci:

$tab[0] \leq tab[1] \leq \dots \leq tab[n-1]$

1. Se compară elementul din mijlocul tabloului (de indice  $((n-1)/2)$ ) cu  $a$ .
2. Dacă acesta are valoarea egală cu  $a$ , înseamnă că s-a găsit un element de valoare  $a$  și se returnează indicele lui.

Altfel se continuă cu punctul 3.

3. Dacă elementul din mijlocul tabloului este diferit de  $a$ , atunci din cauză că tabloul este sortat, elementul căutat se află fie în prima jumătate a tabloului, fie în a doua jumătate.

Mai mult decât atât, se poate stabili exact în care jumătate urmează a se face căutarea și anume:

- Dacă  $tab[(n-1)/2] > a$ , atunci elementul căutat se poate afla numai în prima jumătate a tabloului  $tab$ . Deoarece elementele tabloului următoare lui  $tab[(n-1)/2]$ , sînt mai mari decât el, deci urmează a face căutarea în subtabloul:

$$\text{tab}[0] \leq \text{tab}[1] \leq \dots \leq \text{tab}[(n-1)/2-1].$$

- Dacă  $\text{tab}[(n-1)/2] < a$ , atunci elementul căutat se află numai în jumătatea a doua a tabloului *tab*. Deoarece elementele tabloului, precedente lui  $\text{tab}[(n-1)/2]$ , sînt mai mici decît el, deci căutarea se va face în subtabloul:

$$\text{tab}[(n-1)/2+1] \leq \text{tab}[(n-1)/2+2] \leq \dots \leq \text{tab}[n-1].$$

4. După determinarea subtabloului în care trebuie căutat elementul de valoare *a*, se procedează în mod analog, adică se determină elementul aflat la mijlocul acestui subtablou și apoi se procedează ca la pașii 2 și 3 de mai sus și așa mai departe.

**Exemplu:**

Fie tabloul *tab* de 10 elemente care au valorile:

$\text{tab}[0]=3$	$\text{tab}[1]=4$	$\text{tab}[2]=7$
$\text{tab}[3]=10$	$\text{tab}[4]=13$	$\text{tab}[5]=20$
$\text{tab}[6]=21$	$\text{tab}[7]=35$	$\text{tab}[8]=41$
$\text{tab}[9]=48$		

Se cere indicele elementului de valoare 20.

Avem:

$$n=10, [(n-1)/2]=[9/2]=4.$$

Se compară  $\text{tab}[4]$  cu 20:

$$\text{tab}[4]=13 < 20$$

deci pentru căutarea următoare se alege jumătatea a doua a tabloului:

$$\text{tab}[5]=20, \text{tab}[6]=21, \text{tab}[7]=35, \text{tab}[8]=41, \text{tab}[9]=48.$$

Elementul din mijlocul acestui tablou este de indice  $[(5+9)/2]=7$ . Avem:

$$\text{tab}[7]=35 > 20$$

deci se va face căutarea în prima jumătate a acestui subtablou, adică se selectează subtabloul:

$$\text{tab}[5]=20, \text{tab}[6]=21$$

În continuare, se alege elementul de indice  $[(5+6)/2]=5$ , deci  $\text{tab}[5]$ . Cum  $\text{tab}[5]=20$ , rezultă că în acest moment a fost găsit elementul căutat și se revine din funcție cu valoarea 5 (indicele elementului căutat).

Pentru determinarea, la fiecare pas, a subtabloului în care să se facă căutarea este bine să utilizăm două variabile *inf* și *sup*, prima avînd ca valoare indicele primului element al subtabloului curent selectat, iar cea



de-a doua, indicele ultimului element al aceluiași subtablou. Pașii procesului de căutare vor fi:

1.  $inf=0$

Indicele primului element al tabloului.

2.  $sup=n-1$

Indicele ultimului element al tabloului.

3.  $i = (inf+sup)/2$

Indicele elementului din mijlocul tabloului.

4. Dacă

$tab[i]=a$

căutarea se termină și se returnează valoarea lui  $i$ .

Altfel se continuă cu 5.

5. Dacă

$tab[i] > a$

atunci

$sup=i-1$

Limita superioară se coboară la valoarea  $i-1$  pentru a selecta prima jumătate a tabloului.

Se reia de la punctul 3.

Altfel înseamnă că  $tab[i] < a$  și atunci

$inf=i+1$

Limita inferioară crește la valoarea  $i+1$  pentru a selecta jumătatea a doua a tabloului.

Se reia de la punctul 3.

Acest proces se desfășoară corect în cazul în care, în tablou, există cel puțin un element de valoare  $a$ . În caz contrar, el continuă indefinit. Se observă că la pasul 5 de mai sus, una din limite se modifică și anume, dacă se modifică limita  $sup$ , atunci aceasta *descrește*, iar dacă se modifică limita  $inf$ , atunci aceasta *crește*. Inițial  $inf \leq sup$ .

Deoarece la fiecare pas la care nu s-a găsit elementul căutat, una dintre variabilele  $inf$  sau  $sup$  se modifică și  $inf$  crește, iar  $sup$  descrește, rezulta că inegalitatea de mai sus nu va mai fi satisfăcută după un număr finit de pași,

adică se ajunge ca să fie satisfăcută relația:

$inf > sup$

Aceasta se întâmplă când în tablou nu există un element de valoare  $a$ . Deci procesul de calcul indicat mai sus se întrerupe fie la găsirea elementului căutat, fie când se ajunge ca  $inf$  să-l depășească pe  $sup$

Să reluăm exemplul de mai sus pentru  $a=22$ . Și în acest caz se va ajunge la subtabloul:

$tab[5]=20, tab[6]=21$

și

$inf=5, sup=6$

La pasul 3 se determină:

$i=\lfloor(5+6)/2\rfloor=5$

Apoi se ajunge la pasul 5, deoarece

$tab[5] \neq 22$ .

Cum  $tab[5]=20 < 22$ , rezultă că  $inf=i+1=5+1=6$ . Se revine la pasul 3 unde se atribuie lui  $i$  valoarea:

$i=\lfloor(inf+sup)/2\rfloor=\lfloor(6+6)/2\rfloor=6$

Se ajunge din nou la pasul 5, deoarece

$tab[6] \neq 22$ .

Cum  $tab[6]=21 < 22$ , se modifică din nou  $inf$ :

$inf=i+1=7$

În acest moment  $inf > sup$  și deci procesul de calcul se întrerupe, concluzia fiind aceea că tabloul respectiv nu conține un element de valoare  $a$ .

## FUNCȚIA BIV42

```
int dcauthin(double tab[], int n, double a)
/* cauta în tab un element de valoare egala cu a;
   returneaza indicele elementului respectiv sau -1, daca nu exista un
   astfel de element */
{
    int i, inf, sup;

    inf = 0;
```

```

sup = n-1;
while( inf <= sup ) {
    i = (inf+sup)/2;
    if( tab[i] == a )
        return i;
    if( tab[i] > a )
        sup = i-1;
    else
        inf = i+1;
} /* sfirsit while */

/* se ajunge aici cind inf > sup, ceea ce inseamna ca in tab nu exista
un element de valoare a */
return -1;
}

```

- 4.43 Să se scrie un program care citește un șir de întregi separați prin caractere albe și-i afișează pe cei care sînt numere prime și sînt în intervalul (0,1000).

După ultimul număr se tastează un caracter nenumeric.

Acest program generează la început un tablou cu numere prime. În acest scop se realizează o funcție pe care o numim *prim*. Ea are doi parametri: *tprim* și *n*. Primul parametru este un tablou de tip *int*, ale cărui elemente sînt numerele prime care nu-l depășesc pe *n*:

$tprim[0]=1, tprim[1]=2, tprim[2]=3, tprim[3]=5$  etc.

El este sortat crescător și ultimul element  $tprim[m]$  este cel mai mare număr prim care nu-l depășește pe *n*:

$tprim[m] \leq n$ .

Funcția *prim* returnează numărul numerelor prime păstrate în *tprim* (adică *m*+1).

Pentru determinarea numerelor prime vom proceda ca mai jos:

Primele 3 numere prime (1, 2 și 3) se păstrează în mod automat, dacă  $n > 3$ .

Pentru a determina că un număr  $m > 2$  este prim, este suficient să stabilim că el nu este divizibil cu nici unul din numerele prime mai mici decît el. De fapt, este suficient să testăm divizibilitatea lui cu numerele prime ale căror pătrate nu-l depășesc. De asemenea, vom observa că singurul număr prim par este 2. Deci un număr  $m > 2$  este prim dacă:

1.  $m$  este impar. -
2.  $m$  nu se divide cu nici unul din numerele prime consecutive  $p_1, p_2, \dots, p_k$ , ale căror pătrate nu-l depășesc pe  $m$  ( $p_1 = 3$ ).

Deci  $p_k$  este cel mai mare număr prim pentru care  $p_k * p_k < m$ .

Dacă tabloul conține deja  $i$  numere prime:

$tprim[0], tprim[1], \dots, tprim[i-1]$  ( $i > 2$ )

atunci pentru a determina numărul prim următor vom proceda astfel:

1.  $tprim[i] = tprim[i-1] + 2$ , deoarece numerele prime mai mari decât doi nu sînt pare.
2. Se testează dacă  $tprim[i]$  este prim; în acest scop se caută dacă există printre numerele prime:

$tprim[2] = 3, tprim[3], \dots$

vreunul care să-l dividă pe  $tprim[i]$ .

Conform celor spuse mai sus, nu trebuie încercate toate elementele tabloului *tab* ci numai acele elemente care satisfac relația:

(1)  $tprim[j] * tprim[j] \leq tprim[i]$  ( $j = 2, 3, \dots$ ).

În cazul în care există un element care satisface atît relația (1), cît și relația:

(2)  $tprim[i] \% tprim[j] = 0$

numărul  $tprim[i]$  nu este prim și se va trece la pasul 3 de mai jos

Altfel  $tab[i]$  este prim, se mărește  $i$  cu o unitate și procesul de calcul se reia de la punctul 1.

3. Cum  $tprim[i]$  nu este prim, acesta se mărește cu 2 și procesul se reia de la punctul 2.

Procesul de calcul de mai sus se continuă atît timp cît  $tprim[i] \leq n$

După generarea numerelor prime, programul citește un întreg și dacă acesta aparține intervalului  $(0, 1000)$ , îl caută în tabela de numere prime. Apoi se afișează numărul respectiv însoțit de un mesaj corespunzător. După afișare se revine la citirea numărului următor cu care se procedează analog. Execuția programului se întrerupe la întîlnirea unui caracter care nu aparține unui întreg.

Căutarea numărului citit în tabloul de numere prime se face prin metoda căutării binare. În acest scop se utilizează o funcție analogă cu funcția *dcauthm* definită în exercițiul precedent. În acest caz se va folosi funcția

*icautbin*. Diferența între cele două funcții constă în aceea că *dcautbin* caută o valoare de tip *double* într-un tablou de tip *double*, iar *icautbin* caută o valoare de tip *int* într-un tablou de tip *int*.

### PROGRAMUL BIV43

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 1000

int icautbin(int tab[], int n, int a)
/* cauta in tab un element de valoare egala cu a; returneaza indicele
   elementului respectiv sau -1, daca nu exista un astfel de element */
{
    int i, inf, sup;

    inf = 0;
    sup = n-1;
    while(inf <= sup) {
        i = (inf+sup)/2;
        if(tab[i] == a)
            return i;
        if(tab[i] > a)
            sup = i-1;
        else
            inf = i+1;
    } /* sfirsit while */

    return -1;
} /* sfirsit icautbin */

int prim( int tprim[], int n)
/* pastreaza in tprim numerele prime care nu-l depasesc pe n */
{
    int i, j, k;

    tprim[0] = 1;
    if( n <= 1 )
        return 1;
    tprim[1] = 2;
    if( n <= 2 )
        return 2;
```

```

tprim[2] = 3;
if( n <= 3)
    return 3;
tprim[3] = 5;
i = 3;

while(tprim[i] <= n) {
/* se cauta existenta unui divizor prim >= 3 pentru tprim[i] */
    j = 2;
    while((k=tprim[j]*tprim[j]) < tprim[i] &&
          tprim[i]%tprim[j])

/* ciclul continua atit timp cit tprim[j]*tprim[j] < tprim[i] si restul
   impartirii lui tprim[i] la tprim[j] este diferit de zero */
        j++;

/* ciclul while se termina cind cel putin una din conditiile de mai sus nu
   este adevarata */
    if( k > tprim[i] ) {

/* tprim[i] este prim deoarece nu exista un divizor prim a lui tprim[i] al
   carui patrat sa fie mai mic sau egal cu tprim[i] */
        i++;
        tprim[i] = tprim[i-1] + 2;

/* tprim[i-1] fiind prim, urmatorul numar prim va fi cel putin cu 2 mai
   mare decit el */

    } /* sfirsit if pentru tprim[i] prim */

    else /* tprim[i] nu este prim deoarece s-a determinat un j
          asa incit
              tprim[j]*tprim[j] = tprim[i]
          sau
              restul impartirii lui tprim[i] la tprim[j] este zero;
          in ambele cazuri tprim[i] nu este prim;
          se mareste cu 2 */

        tprim[i] += 2;

    } /* sfirsit while exterior */
return i;

```

```

} /* sfirsit prim */

main() /* citește un sir de intregi si-i afisează pe cei din intervalul
      (0,1000), care sînt numere prime */
{
    int nrprime[MAX];
    int nrcrt;
    int n;

    n=prim(nrprime,MAX);

    /* n este numărul numerelor prime care nu-l depășesc pe 1000 */
    do {
        printf("Tastați un intreg de cel mult 3\
              cifre:");
        if(scanf("%d", &nrcrt) != 1)
            exit(0);
        if(nrcrt == 1 || nrcrt == 2 || nrcrt == 3 ) {
            printf("%d este un numar prim din\
                  intervalul (0,1000)\n", nrcrt);
            continue;
        }
        if(nrcrt>3 && nrcrt<1000 &&
           icautbin(nrprime,n,nrcrt) != -1)
            printf("%d este un numar prim din\
                  intervalul (0,1000)\n", nrcrt);
        else
            printf("%d nu este un numar prim din\
                  intervalul (0,1000)\n", nrcrt);
    } while(1);
} /* sfirsit main */

```

**Observație:**

Condiția

$tprim[j] * tprim[j] < tprim[i]$

poate fi înlocuită cu echivalenta ei:

$tprim[j] < \sqrt{tprim[i]}$

care este mai eficientă pentru numere relativ mari, dacă se calculează rădăcina pătrată în afara ciclului *while* interior:

```

...
while (tprim[i] <= n){

```

```
j=2;  
k=sqrt(tprim[i]);  
while(tprim[j] < k && tprim[i] % tprim[j])  
    j++;  
if(tprim[j] > k){  
    ...
```



## 5. CLASE DE MEMORIE

La compilarea unui text sursă, compilatorul alocă memorie pentru variabilele programului. Se pot alocă variabile pe stivă, în regiștrii calculatorului, cit și în alte zone de memorie.

Programatorul definește, cu ajutorul declarațiilor, modul de alocare al variabilelor. Se pot defini variabile utilizabile (*vizibile*) în *tot* programul, precum și variabile care au utilizări *locale* (cu vizibilitate *restrinsă*). Variabilele care pot fi utilizate în tot programul se numesc variabile *globale*, iar cele care pot fi utilizate numai într-o anumită parte a programului se numesc variabile *locale*.

### 5.1. Variabile globale

Variabilele globale au o *definiție* și eventual una sau mai multe declarații de variabilă *externă*

Definiția unei variabile globale coincide cu o declarație obișnuită care însă este scrisă în afara corpului oricărei funcții a programului.

O astfel de definiție, de obicei, se scrie la începutul unui fișier sursă. Aceasta deoarece ea este valabilă din locul în care este scrisă și pînă la sfîrșitul fișierului sursă respectiv.

În cazul în care programul se compune din mai multe fișiere sursă, o variabilă globală poate fi utilizată într-un fișier sursă în care nu este definită dacă este declarată ca variabilă externă în acel fișier. O declarație de *variabilă externă* coincide cu o declarație de variabilă obișnuită precedată de cuvîntul cheie *extern*.

#### Exemplu:

Un program se compune din două fișiere sursă, *fis1.cpp* și *fis2.cpp*. În fișierul *fis1.cpp* se definesc variabilele globale *zi*, *luna* și *an* de tip *int*. Aceste variabile se utilizează atît în *fis1.cpp*, cit și în *fis2.cpp*. Dacă aceste fișiere se compilează separat, folosind un fișier de tip *proiect* sau dacă *fis2.cpp* se include în fișierul *fis1.cpp* înainte de a defini variabilele globale respective, atunci ele nu sînt definite în *fis2.cpp* și în consecință nu pot fi utilizate în funcțiile din fișierul *fis2.cpp*. Pentru a elimina acest neajuns, vom declara variabilele respective ca externe în funcțiile din fișierul *fis2.cpp*

### Fișierul fis1.cpp

```
int zi, luna, an; /*Definitie de variabile globale.  
                 Aceste variabile pot fi folosite in toate  
                 functiile din fisierul fis1.cpp care  
                 urmeaza după aceasta definitie */  
  
...  
tip f(...)  
{  
    int i;  
    ...  
  
    i=an % 4==0 && an % 100 != 0 || an % 400==0;  
    ...  
  
    if(luna==2)  
        zi=28+i;  
    ...  
}  
  
tip g(...)  
{  
    ...  
    if(zi <= 27)  
        zi++;  
    ...  
    if(zi==31&&luna==12)  
        an++;  
    ...  
}
```

### Fișierul fis2.cpp

```
...  
  
main()  
{  
    extern int zi, luna, an; /* declaratie de extern pentru  
                             variabilele globale zi, luna si an,  
                             definite in fis1.cpp;  
                             este valabila in corpul functiei  
                             main */  
  
    ...  
    if (zi<1 || zi>31)
```

```

        printf("ziua eronata\n");
    if(luna<1 || luna>12)
        printf("luna eronata\n");
    if(an<1600 || an>4900)
        printf("anul eronat\n");
    ...
} /* sfirsit main */

tip p(...)
{
    ... /* nu exista declaratii de extern pentru variabilele
        globale zi,luna,an; in acest caz nu se pot utiliza variabilele
        respective numai daca fis2.cpp se include in fis1.cpp
        dupa definitia acestor variabile globale */
    ...
    zi=15; /* zi apare la compilare ca nedefinita */
    ...
}

```

#### Observație:

Întrucît repartizarea funcțiilor în fișiere poate să se modifice pe măsură ce se construiește programul, se recomandă ca orice utilizare a unei variabile globale să fie precedată de declarația de *extern* a ei.

Variabilele globale sînt alocate în momentul compilării. Lor li se alocă memorie, la întîlnirea definițiilor lor, într-o zonă de memorie prevăzută special pentru variabilele globale ale programului.

## 5.2. Variabile locale

Variabilele locale, spre deosebire de cele globale, nu sînt valabile în tot programul. Ele au o valabilitate locală, în unitatea în care sînt declarate.

Variabilele locale pot fi alocate pe *stivă*. În acest caz ele se numesc *automatice*. Acestea se declară în mod obișnuit, în corpul unei funcții sau la începutul unei instrucțiuni compuse. O astfel de variabilă se alocă la execuție (nu la compilare).

La apelul unei funcții, variabilele automate (declarate în mod obișnuit înaintea primei instrucțiuni din corpul funcției respective) se alocă pe stivă. În momentul în care se revine din funcție, variabilele automate alocate la apel, se dezalocă (elimină) și stiva revine la starea dinaintea apelului (operația de curățire a stivei). Aceasta înseamnă că variabilele automate își pierd existența la revenirea din funcția în care sînt declarate. De aceea, o

variabilă automată este valabilă (vizibilă) numai în corpul funcției în care a fost declarată.

În același mod se comportă variabilele automate declarate la începutul unei instrucțiuni compuse. O astfel de variabilă se alocă pe stivă în momentul în care controlul programului ajunge la instrucțiunea compusă în care este declarată variabila respectivă și se elimină de pe stivă în momentul în care controlul programului trece la instrucțiunea următoare celei compuse.

Variabilele locale pot și să nu fie alocate pe stivă. În acest scop ele se declară ca fiind *statice*. O declarație de variabilă statică este o declarație obișnuită precedată de cuvântul cheie *static*.

Variabilele statice pot fi declarate atât în corpul unei funcții cit și în afara corpului oricărei funcții.

O variabilă statică declarată în corpul unei funcții este definită numai în corpul funcției respective.

Spre deosebire de variabilele automate, o variabilă statică nu se alocă pe stivă la execuție, ci la compilare într-o zonă de memorie destinată acestora.

O variabilă statică declarată în afara corpurilor funcțiilor este definită (vizibilă) din punctul în care este declarată și pînă la sfîrșitul fișierului sursă care conține declarația respectivă. Spre deosebire de variabilele globale, o astfel de variabilă nu poate fi declarată ca externă. Deci ea nu poate fi utilizată în alte fișiere dacă acestea se compilează separat sau se includ înaintea declarației respective.

Putem spune că o variabilă statică declarată în afara corpurilor funcțiilor este locală fișierului sursă în care este declarată.

Ea se alocă la compilare într-o zonă specială rezervată variabilelor statice corespunzătoare fișierului sursă în care au fost declarate.

Dacă prin *modul* înțelegem un text sursă compilat separat față de restul textului sursă al unui program, atunci putem afirma că o variabilă statică declarată în afara corpurilor funcțiilor este locală modulului în care a fost declarată. La aceasta mai trebuie să adăugăm și faptul că utilizările unei variabile statice trebuie să fie precedate de declarația ei.

În cazul variabilelor globale, acestea pot fi utilizate în orice modul, dacă sînt precedate, în modulul respectiv, fie de definiția lor, fie de declarația de extern.

Funcțiile sînt analoge cu variabilele globale. Ele, ca și variabilele globale, pot fi utilizate (apelate) în orice modul al programului dacă apelul lor este

precedat, în modulul respectiv, fie de definiție, fie de prototipul lor. În acest caz, prototipul funcției înlocuiește declarația de *extern* a variabilei globale.

Limbajele C și C++ permit declarări de funcții statice. Aceasta se face precedind antetul funcției de cuvântul cheie *static*. În acest caz, funcția respectivă este locală modulului în care este definită.

Împărțirea pe module a unui program nu se face prin reguli definite riguros.

Se obișnuiește să se spună că un modul este format din funcții "înrudite" și conține date pe care acestea le prelucrează în comun.

#### Exemplu:

Un program se compune din 2 fișiere *sursa1.cpp* și *sursa2.cpp* care formează fiecare câte un modul (se compilează separat, utilizând, de exemplu, un fișier de tip *project*).

#### Fișierul *sursa1.cpp* conține:

```
int i, j, k; /* - definitie de variabile globale;
               - se pot utiliza in tot modulul;
               - se pot declara ca externe in celalalt modul pentru
                 a putea fi utilizate si acolo */

static double x, y; /* - declaratie de variabile statice;
                       - sint utilizabile in tot modulul de fata
                       si numai in acesta */

...

main()
{
    int p, q; /* - declaratie de variabile automate;
                - sint utilizabile numai in corpul functiei main */

    static int u, v, r; /* - declaratie de variabile statice;
                          - sint utilizabile numai in corpul functiei
                          main */

    ...
    /* se pot utiliza variabilele i, j, k, x, y, p, q, u, v si r */
    ...
}
```

```

tip f(...)
{
    float a,b; /*- declaratie de variabile automate;
               - sint utilizabile numai in corpul functiei f */

    static char c,temp; /*- declaratie de variabile statice;
                        - sint utilizabile numai in corpul
                        functiei f */

    ...

    /* se pot utiliza variabilele i, j, k, x, y, a, b, c si temp */
}

```

**Fișierul sursa2.cpp conține:**

```

static float a,t; /*- declaratie de variabile statice;
                  - sint utilizabile in fisierul sursa2.cpp */

tip g(...)
{
    extern int i,j,k; /*- declaratie de extern pentru variabilele
                      i,j,k;
                      - sint utilizabile in corpul functiei g */

    char d,e; /*- declaratie de variabile automate;
               - sint utilizabile in corpul functiei g */

    static double d1,d2,d3; /*- declaratie de variabile statice;
                            - sint utilizabile in corpul
                            functiei g */

    ...

    /* se pot utiliza variabilele i, j, k, s, t, d, e, d1, d2 si d3 */
    ...
}

tip h(...)
{
    extern int k; /*- declaratie de extern pentru variabila k;
                  - este utilizabila in corpul functiei h */

    ...
}

```

```
/* se pot utiliza variabilele k, s si t */
...
}
```

Variabilele pot fi *redeclarate* indiferent de modul lor de alocare (global, static sau automatic).

**Exemplu:**

```
int i,c; /* variabile globale */
static float a; /* variabila statica */
...
main()
{
    double c; /* variabila automatica diferita de variabila globala c */
    char a; /* variabila automatica diferita de variabila statica a */
    ...
    c=123; /* se atribuie variabilei automate c valoarea 123 dupa
           conversia ei in flotanta dubla precizie */
    ...
    i=123; /* se atribuie variabilei globale i valoarea 123 */
    ...
    a='1'; /* se atribuie variabilei automate a codul ASCII
           al caracterului 1 */

    /* - in aceasta functie nu se pot utiliza variabila globala c si variabila
       statica a deoarece acestea au fost redeclarate;
       - cu toate acestea, in limbajul C++ este posibila utilizarea lor
       folosind operatorul :: pentru globale. */
}

tip f(...)
{
    ...
    /* nu sînt redeclaratii pentru i,c si a */

    c=123; /* se atribuie variabilei globale c valoarea 123 */
    a=123; /* se atribuie variabilei statice a valoarea 123, dupa
           ce a fost convertita spre flotant simpla precizie */
    ...
}
```

Uneori, tablourile de dimensiuni mari se alocă globale sau statice

deoarece alocarea lor pe stivă poate conduce la depășirea stivei.

## 5.3. Alocarea parametrilor

Parametrii formali se alocă pe stivă ca și variabile automate. De aceea, ei se consideră a fi variabile locale și sint utilizabili numai în corpul funcției în anietul căreia sint declarați.

La apelul unei funcții, se alocă pe stivă parametri formali, dacă există, li se atribuie valorile parametrilor efectivi care le corespund. Apoi se alocă pe stivă variabilele automate declarate la începutul corpului funcției respective.

La revenirea din funcție, se realizează curățirea stivei, adică sint eliminate de pe stivă (dezalocate) atit variabilele automate, cit și parametri. În felul acesta, la revenirea din funcție, stiva ajunge la starea dinaintea apelului.

## 5.4. Utilizarea parametrilor și a variabilelor globale

Am văzut că parametri efectivi, se utilizează pentru a concretiza valorile care intervin în procesul de calcul definit printr-o funcție. Se obișnuiește să se spună că valorile respective se "transferă" funcției prin intermediul parametrilor.

Din această cauză, parametri se mai consideră ca fiind o "interfață" între funcții. În limbajul C această interfață este unilaterală, adică ea transferă date de la funcția care face apelul, la funcția apelată.

Variabilele globale reprezintă și ele o interfață între funcții. În acest caz nu se transferă date de la o funcție la alta, ci orice funcție are acces la valorile variabilelor globale. Dacă o funcție trebuie să transtere o valoare funcției apelate, atunci aceasta are două posibilități:

1. Să folosească un parametru la care i se atribuie valoarea respectivă înainte de apel.
2. Să atribuie valoarea respectivă unei variabile globale înainte de apel.

În primul caz, funcția apelată are destinat un parametru formal la care i se atribuie în mod automat valoarea respectivă la apel. În al doilea caz, funcția apelată trebuie să aibă acces la variabila globală căreia i s-a atribuit valoarea de transferat. În acest scop, este necesar să se cunoască numele variabilei globale respective în momentul în care se programează funcția



respectivă. Aceasta este un neajuns al variabilelor globale față de parametri. Numele și sensul variabilelor globale trebuie să fie cunoscut înainte de a începe programarea diferitelor funcții care utilizează în comun variabilele respective.

Un alt dezavantaj al variabilelor globale este sursa mare de erori pe care o reprezintă. Într-adevăr, orice funcție din program avînd acces la o variabilă globală, ea poate modifica valoarea acesteia și de aceea posibilitatea modificării eronate este mare. De asemenea, depistarea unei astfel de erori este destul de complicată, deoarece este necesar să se studieze un număr mare de funcții (toate funcțiile care au acces la variabila globală a cărei valoare s-a constatat că este eronată).

În cazul utilizării parametrilor, se realizează o protecție a datelor, deoarece funcția apelată nu poate modifica valorile parametrilor efectiv în mod direct.

Avantajul variabilelor globale decurge din faptul că ele reprezintă o interfață simplă între funcții, interfață care este în ambele sensuri. O funcție poate modifica valoarea unei variabile globale, modificare care rămîne valabilă la revenirea din ea, deci funcția care a făcut apelul poate beneficia de modificarea făcută prin intermediul funcției apelate. Dar chiar acest fapt se consideră că este o sursă de erori. De aceea, se consideră că nu este bine să se exagereze cu utilizarea variabilelor globale, ele constituind o sursă de erori.

De obicei, folosim variabilele globale cînd rezultatele unei funcții sînt folosite în comun de mai multe funcții ale programului. În rest, se recomandă utilizarea parametrilor pentru realizarea interfețelor dintre funcții.

## 5.5. Variabile registru

O variabilă alocată într-un registru al calculatorului se numește *variabilă registru*. Ea se declară printr-o declarație obișnuită precedată de cuvîntul cheie *register*.

Se pot declara, ca variabile registru, numai variabilele de tip *int*, *char* și *pointer* (acest tip va fi introdus într-un capitol următor). De asemenea, numai parametrii și variabilele automate simple pot fi declarate ca variabile registru.

Numai un număr limitat de variabile se pot alocă în regiștri. Alocarea se face la apelul unei funcții și în ordinea în care au fost declarate aceste variabile în funcția respectivă. Alocarea rămîne valabilă pînă la revenirea din funcție. În cazul în care nu pot fi alocate în regiștri toate variabilele

registru declarate de programator, se alocă cite se pot în ordinea declarării lor, iar restul se alocă în mod obișnuit pe stivă, ca orice variabilă automatică sau parametru.

Se recomandă să se declare ca variabile registru, acele variabile ale unei funcții care au o utilizare mare în funcția respectivă. Alocarea unei variabile în registru conduce la economie de memorie, precum și la creșterea vitezei de calcul.

Menționăm că, în lipsa declarațiilor registru, compilatorul alocă în mod implicit anumite variabile automate sau parametri în regiștri.

În legătură cu variabilele registru amintim că lor nu li se poate aplica operatorul adresă (&).

### Exerciții:

5.1 Să se scrie o funcție care citește:

- valoarea variabilei  $m$  de tip `int`;
- valoarea variabilei  $n$  de tip `int`;
- $m \cdot n$  numere care reprezintă elementele unei matrice de ordinul  $m \cdot n$ .

Funcția are următorii parametri:

<i>dmat</i>	- Tablou care păstrează elementele matricei citite.
<i>max</i>	- Maximul produsului $m \cdot n$ admis.

Funcția returnează produsul  $m \cdot n$  și atribuie valorile lui  $m$  și  $n$ , variabilelor globale *nrlin*, respectiv *nrcol*.

Tabloul *dmat* este unidimensional, matricea păstrându-se prin *liniarizare*.

Dacă notăm cu  $a[i,j]$ ,  $i=0,1,\dots,m-1$ ;  $j=0,1,\dots,n-1$ ; elementele matricei, atunci elementul  $a[i,j]$  se atribuie lui  $dmat[k]$ , unde:

$$(1) \quad k = i \cdot n + j$$

Această relație poartă denumirea de relație de *liniarizare*. Ea permite păstrarea pe linii a elementelor matricei citite. Astfel, prima linie a matricei este formată din elementele  $a[0,j]$ , pentru  $j=0,1,\dots,n-1$ . Conform relației (1), aceste elemente se păstrează în tabloul *dmat* prin intermediul elementelor:

$$dmat[0], dmat[1], \dots, dmat[n-1].$$

Linia a doua, adică elementele  $a[1,j]$ , pentru  $j=0,1,\dots,n-1$ , se păstrează în tabloul *dmat* ca elemente ce au indicii  $k=1 \cdot n + j$ , adică:

$\text{dmat}[n], \text{dmat}[n+1], \dots, \text{dmat}[n+n-1]$ .

În general, linia formată din elementele  $a[i,j]$  pentru  $j=0,1,\dots,n-1$  se păstrează în tabloul *dmat* ca:

$\text{dmat}[i*n], \text{dmat}[i*n+1], \dots, \text{dmat}[i*n+n-1]$

Funcția de față folosește funcția *ndcit* definită în exerciciul 4.37.

## FUNCȚIA BV1

```
int gdcitmat(double dmat[], int max)
/* citește
   m-numar de linii;
   n-numar de coloane;
   m*n numere de tip double pe
   care le pastreaza in matricea dmat prin liniarizare;
   returneaza valoarea m*n;
   realizeaza atribuirile:
   nrlin=m;
   nrcol=n; */
{
    extern int nrlin,nrcol;
    int i,m,n;
    char t[255];

    do { /* se citesc valorile lui m si n */
        do { /* citește pe m */
            printf("numarul de linii= ");
            if(gets(t) == NULL) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(t,"%d", &m) == 1 && m > 0 &&
                m <= max)
                break;
            printf("nu s-a tastat un intreg in\
                intervalul [1,%d]\n",max);
        } while(1);
        do { /* citește pe n */
            printf("numarul de coloane= ");
            if(gets(t) == NULL) {
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(sscanf(t,"%d", &n) == 1 && n > 0 &&
                n <= max)
                break;
            printf("nu s-a tastat un intreg in\
                intervalul [1,%d]\n",max);
        } while(1);
        nrlin=m;
        nrcol=n;
        return m*n;
    }
}
```

```

    }
    if(sscanf(t,"%d",&n) == 1 && n > 0 &&
       n <= max )
        break;
    printf("nu s-a tastat un intreg in\
           intervalul [1,%d]\n",max);
} while (1);
i = m*n;
if(i <=max)
    break;
printf("produsul m*n=%d depaseste pe\
       max=%d\n", i, max );
printf("se reiau citirile lui m si n\n");
} while(1);

/* se citesc elementele matricei tastate pe linii */
if(ndcit(i,dmat) != i ) {
    printf("nu s-au tastat %d elemente\n", i);
    exit(1);
}
nrlin = m;
nrcol = n;
return i;
}

```

5.2 Să se scrie o funcție care calculează produsul dintre o matrice de ordinul  $m \times n$  și o matrice coloană (de ordinul  $n \times 1$ ).

Funcția are următorii parametrii:

<i>m</i>	- Numărul liniilor matricei.
<i>n</i>	- Numărul coloanelor matricei.
<i>dmat</i>	- Tablou unidimensional de tip <i>double</i> care păstrează elementele matricei de ordinul $m \times n$ prin liniarizare.
<i>dmatcol</i>	- Tablou unidimensional de tip <i>double</i> care păstrează elementele matricei coloana.
<i>dmatrez</i>	- Tablou unidimensional de tip <i>double</i> care păstrează elementele matricei rezultat.

Matricea rezultat este de ordinul  $m \times 1$ . Elementele ei se calculează prin relația de mai jos:

$$dmatrez[i] = a[i,0] * dmatcol[0] + a[i,1] * dmatcol[1] + \dots + a[i,n-1] * dmatcol[n-1]$$

pentru  $i=0,1,\dots,m-1$ , unde prin  $a[i,j]$  am notat un element al matricei de ordinul  $m \times n$ . Ținând seama de relația de liniarizare, suma de mai sus se scrie astfel:

$$\begin{aligned} \text{dmatrez}[i] &= \text{dmat}[i \cdot n] \cdot \text{dmatcol}[0] + \\ &\text{dmat}[i \cdot n + 1] \cdot \text{dmatcol}[1] + \dots + \text{dmat}[i \cdot n + n - 1] \cdot \text{dmatcol}[n - 1]. \end{aligned}$$

## FUNCȚIA BV2

```
void pmatcol(int m, int n, double dmat[],
             double dmatcol[], double dmatrez[])
```

```
/* calculeaza produsul dintre matricea dmat de ordinul m*n si
   vectorul coloana dmatcol de ordinul n*1; se obtine matricea
   coloana dmatcol de ordinul m*1 */
```

```
{
    int i, j, k;

    for(i=0; i < m; i++) {
        dmatrez[i] = 0;
        k=i*n;
        for(j=0; j <= n-1; j++)
            dmatrez[i] += dmat[k+j] * dmatcol[j];
    }
}
```

- 5.3 Să se scrie un program care citește elementele unei matrice  $a$  de ordinul  $m \times n$ , elementele unei matrice coloane  $b$  de ordinul  $n \times 1$ , calculează și afișează produsul  $a \cdot b$ .

În acest scop se vor apela trei funcții:

- funcția *gdcumat* care citește elementele matricei  $a$  (vezi exercițiul 5.1.),
- funcția *dvcut* care citește elementele matricei  $b$  (vezi exercițiul 4.38.);
- funcția *pmatcol* care calculează produsul  $a \cdot b$  (vezi exercițiul 5.2.).

Deoarece funcțiile *gdcumat* și *dvcut* apelează funcția *ndcut*, se va include și fișierul care conține definiția acestei funcții (vezi exercițiul 4.37).

Matricea  $a$  va fi precedată de doi întregi care reprezintă valorile lui  $m$  și  $n$ .

Matricea  $b$  va fi precedată de un întreg care reprezintă valoarea lui  $n$ . Această valoare, deși pare că nu este necesară înaintea elementelor matricei  $b$ , ea se tastează pe de o parte pentru a face o verificare cu privire la numărul

datelor tastate, iar pe de altă parte, prezența ei este cerută de funcția *dvcit*.

## PROGRAMUL BV3

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "biv38.cpp" /* functia dvcit */
#include "bv1.cpp" /* functia gdcitmat */
#include "bv2.cpp" /* functia pmatcol */

int nrlin,nrcol; /* variabile globale utilizate la citirea matricei
                  cu ajutorul functiei gdcitmat */

#define MAX 1000

main() /* citeste elementele matricei a de ordinul m*n si elementele
        matricei b de ordinul n*1, calculeaza si afiseaza produsul
        c = a*b */
{
    int m;
    double a[MAX],b[MAX],c[MAX];

    gdcitmat(a,MAX); /* citeste matricea a; la revenire, nrlin are
                       ca valoare numarul liniilor (m), iar nrcol
                       numarul coloanelor(n) */

    /* citeste matricea b */
    if(dvcit(MAX,b) != nrcol) {
        printf("matricea coloana nu are\
              ' %d linii\n",nrcol);
        exit(1);
    }

    /* calculeaza produsul c = a*b */
    pmatcol(nrlin,nrcol,a,b,c);

    /* listeaza matricea rezultat */
    for( m = 0; m < nrlin; m++) {
        printf("c[%d]=%g\n",m,c[m]);
        if((m+1)%23 == 0) {
            printf("actionati o tasta pentru a\
                  continua\n");
        }
    }
}
```

```

        getch();
    }
}

```

5.4 Să se scrie o funcție care calculează produsul a două matrice cu elemente de tip *double*.

Fie  $a$  o matrice de ordinul  $m \times n$  și  $b$  o matrice de ordinul  $n \times s$ . Matricea:

$c = a \cdot b$

este de ordinul  $m \times s$ . Elementele ei se calculează folosind relația:

$$c[i,j] = a[i,0] \cdot b[0,j] + a[i,1] \cdot b[1,j] + \dots + a[i,n-1] \cdot b[n-1,j]$$

pentru  $i=0,1,2,\dots,m-1$  și  $j=0,1,2,\dots,s-1$ .

Elementele celor trei matrice se păstrează liniarizat în tablourile unidimensionale *dmata*, *dmatb* și *dmatac*. Folosind relația de liniarizare, suma de mai sus se reprezintă astfel:

$$\begin{aligned} \text{dmatac}[i \cdot s + j] = & \text{dmata}[i \cdot n] \cdot \text{dmatb}[j] + \\ & \text{dmata}[i \cdot n + 1] \cdot \text{dmatb}[s + j] + \dots + \\ & \text{dmata}[i \cdot n + k] \cdot \text{dmatb}[k \cdot s + j] + \dots + \\ & \text{dmata}[i \cdot n + n - 1] \cdot \text{dmatb}[(n - 1) \cdot s + j] \end{aligned}$$

## FUNCȚIA BV4

```

void dprodmat(int m,int n,int s, double dmata[],
              double dmatb[], double dmatc[])
/* calculeaza produsul matricelor a si b:
   c = a*b;
   dmata pastreaza matricea a de ordinul m*n,
   dmatb pastreaza matricea b de ordinul n*s;
   dmatc pastreaza matricea c de ordinul m*s */
{
    int i,j,k;
    int in,is;

    for(i=0; i < m ; i++) {
        is=i*s;
        in= i*n;
        for(j=0; j<s; j++) {
            dmatc[is+j] = 0;
            for(k=0;k<n;k++)
                dmatc[is+j] += dmata[in+k]*dmatb[k*s+j];
        }
    }
}

```

```

    }
}
}

```

5.5 Să se scrie un program care citește elementele a doua matrice de tip *double*, calculează și afișează produsul lor

Programul utilizează funcția *gdcitmat* pentru a citi elementele celor două matrice. De aceea, elementele matricelor vor fi precedate de ordinul lor (numărul de linii și numărul coloanelor - vezi exercițiul 5.1).

## PROGRAMUL BV5

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "bv37.cpp" /* funcția ndcit */
#include "bv1.cpp" /* funcția gdcitmat */
#include "bv4.cpp" /* funcția dprodmat */

#define MAX 1000

int nrlin,nrcol;

main() /* citește elementele a doua matrice, calculează și
        afișează matricea produs */
{
    int m,n,s;
    double a[MAX],b[MAX],c[MAX];
    int i,j,is,k;

    /* citește matricea a */
    gdcitmat(a,MAX);
    m=nrlin; /* numărul liniilor matricei a */
    n=nrcol; /* numărul coloanelor matricei a */

    /* citește matricea b */
    gdcitmat(b,MAX);
    if(nrlin != n) {
        printf("matricea a are %d coloane\n",n);
        printf("matricea b are %d linii\n",nrlin);
        exit(1);
    }
}

```



```

s=nrcol; /* nr coloanelor matricei b */

/* se realizeaza produsul c = a*b */
dprodmat(m,n,s,a,b,c);

/* afiseaza elementele matricei produs */
k=0;
for(i=0;i<m;i++) {
    printf("\n\tlinia nr.%d\n",i+1);
    is=i*s;
    k++;
    for(j=0;j<s;j++) {
        printf("%g ", c[is+j]);
        if(j%5 == 4) {

/* afiseaza 5 elemente pe un rind */

            printf("\n");
            k++; /* numara rindurile */
        }
        if(k==23) {
            printf("actionati o tasta pentru a\
continua\n");
            getch();
            k=1;
        }
    }
}
}
}

```

## 6. INIȚIALIZARE

Adesea dorim ca unele variabile din program să aibă valori inițiale. Acest lucru este posibil și anume, variabilelor globale li se pot da valori inițiale la definirea lor, iar celelalte clase de variabile pot fi inițializate la declararea lor.

### 6.1. Inițializarea variabilelor simple

O variabilă simplă poate fi inițializată printr-o construcție de formă:

*tip nume=expresie;*

sau

*static tip nume=expresie;*

dacă variabila este statică.

Prima formă corespunde variabilelor automate sau globale.

Variabilelor globale sau statice li se atribuie valori inițiale la compilare, deci ele au valorile respective la lansarea programului. Faptul că aceste variabile se inițializează la compilare, cere ca expresiile utilizate pentru inițializare să fie constante.

Variabilele automate simple se inițializează la execuție, de fiecare dată când ele se alocă pe stivă, adică la apelul funcției care conține declarațiile lor. În acest caz, expresiile utilizate pentru inițializare pot să nu fie constante. Cu toate acestea, operandii variabili ai unei astfel de expresii trebuie să aibă valori definite în prealabil. În caz contrar, expresia de inițializare nu poate fi evaluată în momentul în care se alocă pe stivă variabila automată pe care o inițializează.

Variabilele globale și statice neinițializate au în mod implicit valoarea zero. Acest lucru nu mai are loc în cazul variabilelor automate.

O variabilă automată neinițializată are o valoare imprevizibilă în momentul apelului funcției în al cărui corp este declarată. Aceasta rezultă din faptul că ea se alocă pe stivă și în lipsa expresiei de inițializare în declarația sa, ei nu i se atribuie nici o valoare. În felul acesta, valoarea ei rămâne nedefinită până în momentul în care i se atribuie o valoare printr-o instrucțiune de atribuire.

În toate cazurile se fac conversii dacă tipul expresiei de inițializare nu coincide cu tipul variabilei pe care o inițializează.

### Exemple:

1

```
int n=100; /*- definitie de variabila globala;  
- lui n i se atribuie valoarea 100;  
- la lansarea programului n are valoarea 100. */
```

2

```
double x=3; /*- definitie de variabila globala;  
- lui x i se atribuie valoarea 3 dupa ce se convertește  
spre double;  
- la lansarea programului x are valoarea 3. */
```

3.

```
#define MAX 100  
  
int i = MAX*2; /*- i este variabila globala initializata cu  
valoarea 200;  
- la lansarea programului i are valoarea 200. */
```

4.

```
int k; /*- definitie de variabila globala;  
- la lansarea programului k are valoarea zero. */
```

5.

```
static char c='a'; /*- declaratie de variabila statica;  
- lui c i se atribuie codul ASCII al ca-  
racterului a(97);  
- la lansarea programului c are valoarea 97. */
```

6

```
static int s; /*- declaratie de variabila statica;  
- la lansarea programului s are valoarea zero. */
```

7

```
static int p='a'+1; /*- declaratie de variabila statica;  
- la lansarea programului s are ca valoare  
codul ASCII al literei b (97+1). */
```

8.

```
tip f(int n)
{
    int x=123; /* - declaratie de variabila automatica;
                  - la fiecare apel al functiei f, x se alocă
                  pe stiva si i se atribuie valoarea 123. */
    int a=x+n; /* - declaratie de variabila automatica;
                  - la fiecare apel al functiei f, a se alocă
                  pe stiva si i se atribuie valoarea expresiei x+n;
                  - la intalnirea declaratiei ambui operanzi ai
                  expresiei x+n sint definiti. */
}
```

### Exerciții:

6.1 Să se scrie o funcție care are ca parametri doi întregi  $x$  și  $y$ , calculează și returnează numărul aranjamentelor de  $x$  obiecte luate câte  $y$ .

Funcția de față este analogă funcției definite în exercițiul 4.29.

### FUNCȚIA BV11

```
double aranjamente ( int x, int y)
/* calculeaza si returneaza numarul aranjamentelor de
   x obiecte luate cate y */
{
    double a=1.0;
    int i=x-y+1;

    if(x < 1 || x > 170 )
        return -1.0;
    if(y < 1 || y > x )
        return -1.0;
    while( i <= x )
        a *= i++;
    return a;
}
```

### Observație:

Pentru a testa funcția de față se poate utiliza programul din exercițiul 4.30. Programul respectiv apelează funcția *aranjamente* definită în exercițiul 4.29.

Pentru a apela funcția *aranjamente* definită mai sus, în programul din exercițiul 4.30. se va schimba construcția

```
#include "BIV29.CPP"
```

cu

```
#include "BVI1.CPP"
```

## 6.2. Inițializarea tablourilor

Tablourile, ca și variabilele simple, pot fi inițializate. Tablourile *globale* se inițializează prin *definițiile* lor. Tablourile *statice* și *automatice* se inițializează prin *declarațiile* lor.

În limbajul C, în toate cazurile, inițializările se fac prin expresii *constante*.

Tablourile *globale* și *statice* se inițializează la *compilare*. De aceea, la lansarea programului, elementele lor au ca valori, valorile expresiilor cu care au fost inițializate.

Tablourile *automatice* se inițializează la *execuție*, de fiecare dată când se apelează funcția în care sint declarate. Menționam că în unele versiuni ale limbajului C nu se pot inițializa tablourile automate.

Limbajul Turbo C permite inițializarea tablourilor automate.

Inițializarea unui tablou unidimensional se realizează printr-o construcție de forma:

```
tip nume[ec] = {ec0,ec1,...,eci};
```

sau

```
static tip nume [ec] = {ec0,ec1,...,eci};
```

dacă tabloul este static.

Prin *ec*, *ec0*, *ec1*, ..., *eci* am notat expresii constante.

Prin aceste construcții, elementul *nume[k]* se inițializează cu valoarea expresiei constante *eck*, pentru  $k=0,1,2,\dots,i$

În general,  $i \leq ec-1$ . Dacă  $i > ec$ , atunci construcția este eronată. Dacă  $i < ec-1$ , atunci elementele

```
nume[i+1], nume[i+2], ..., nume[ec-1]
```

rămân neinițializate.

Elementele *neinițializate* ale unui tablou *global* sau *static* au în mod implicit valoarea inițială egală cu zero.

Elementele *neinițializate* ale unui tablou *automatic* au valori inițiale *nedefinite*.

În cazul în care se inițializează toate elementele unui tablou unidimensional, se poate omite expresia din parantezele pătrate care definește numărul elementelor tabloului. Deci construcțiile:

```
tip nume[] = {ec0,ec1,...,ecn};
```

și

```
static tip nume[] = {ec0,ec1,...,ecn};
```

sînt corecte și în ambele cazuri tablourile au  $n+1$  elemente, iar elementul

```
nume[i]
```

este inițializat cu valoarea expresiei *eci*.

Menționăm că, la fel ca și în cazul variabilelor simple, dacă expresia de inițializare are un tip diferit de cel al tabloului, atunci valoarea ei se convertește spre tipul tabloului înainte de a fi atribuită elementului pe care-l inițializează.

#### Observație:

Într-o declarație sau definiție de tablou unidimensional se poate omite expresia care definește numărul elementelor tabloului în următoarele cazuri:

1. Definiția sau declarația de tablou conține expresii constante pentru inițializarea fiecărui element al tabloului.
2. Declarația se referă la un tablou unidimensional care este parametru formal.
3. Declarația de tablou extern unidimensional:

```
extern int tab[];
```

#### Exemple:

1. `int tab[10] = {0,1,2,3,4,5,6,7,8,9};`

Prin această definiție/declarație, lui `tab[i]` i se atribuie valoarea *i*.

2. `int tab[] = {0,1,2,3,4,5,6,7,8,9};`

Această construcție este identică, ca efect, cu cea precedentă.

3. `double d[20] = {0,1,-1,3,-2};`

Primele 5 elemente ale tabloului *d* se inițializează astfel:

```

d[0]=0;
d[1]=1;
d[2]=-1;
d[3]=3;
d[4]=-2.

```

Celelalte elemente ale tabloului `d[5]`, `d[6]`, ..., `d[19]` au valoarea inițială zero dacă `d` este un tablou global și o valoare inițială imprevizibilă dacă tabloul este automatic.

4.

```

#define V ('A'-10)
static int chexa[] = {'A'-V, 'B'-V, 'C'-V,
                     'D'-V, 'E'-V, 'F'-V};

```

Elementele tabloului `chexa` se inițializează astfel.

```

chexa[0]='A'-( 'A'-10)=10
chexa[1]='B'-( 'A'-10)=11
chexa[2]=12
chexa[3]=13
chexa[4]=14
chexa[5]=15.

```

5. `char er[] = {'e', 'r', 'o', 'a', 'r', 'e', '\0'};`

Tabloul are 7 elemente care se inițializează astfel:

```

er[0]='e'
er[1]='r'
er[2]='o'
er[3]='a'
er[4]='r'
er[5]='e'
er[6]='\0'

```

Se observă că tabloul `er` păstrează șirul de caractere "eroare", inclusiv caracterul `NUL` care termină orice șir de caractere.

Având în vedere faptul că inițializarea tablourilor de tip `char` se utilizează frecvent, autorii limbajului C au introdus o simplificare pentru inițializarea tablourilor de acest tip. Astfel, pentru un tablou de tip `char` se pot utiliza una din construcțiile:

```

char nume[ec]=șir;

```

sau

**static char nume[ec]=șir;**

unde prin *șir* se înțelege o succesiune de caractere delimitată prin ghilimele. Prin aceste construcții, elementele tabloului *nume* se inițializează cu codurile ASCII ale caracterelor din compunerea șirului de caractere *șir*, iar după ultimul caracter al șirului se memorează caracterul *NUL*. Expresia constantă *ec*, dacă este prezentă, atunci ea este cel puțin egală cu numărul caracterelor proprii ale șirului mărit cu unu.

Din cele de mai sus rezultă că inițializarea tabloului *er* se poate realiza folosind construcția:

**char er[]="eroare";**

Tablourile multidimensionale pot fi și ele inițializate prin construcții analoge. Așa de exemplu, un tablou bidimensional se poate inițializa printr-o construcție de forma:

```
tip nume [n]/[m]={
{ec11,ec12,...,ec1m},
{ec21,ec22,...,ec2m},

{ecn1,ecn2,...,ecnm}
};
```

unde:

*n, m, ecij* - Pentru  $i=1,2,\dots,n$  și  $j=1,2,\dots,m$  sînt expresii constante

Numărul expresiilor constante poate fi mai mic decît *m* în oricare din acoladele corespunzătoare celor *n* linii ale tabloului bidimensional.

În cazul tablourilor statice, declarația este precedată de cuvîntul cheie *static*.

Într-o declarație sau definiție de tablou bidimensional se poate omite numai expresia constantă din prima paranteză pătrată, adică se poate omite numai *n* din construcția de mai sus.

În general, într-o declarație sau definiție de tablou multidimensional se poate omite numai expresia constantă din prima paranteză pătrată, adică limita superioară pentru primul indice. Ea poate fi omisă în una din cele trei cazuri care au fost amintite la tablourile unidimensionale:

- la declarații sau definiții care conțin inițializări,
- la declarația de parametri;
- la declarații de *extern*.



### Exemple:

1.

```
int tab[3][4]={
    {-1,0,1,-1},
    {-1,0,1,2},
    {10,20,30,40}
};
```

Prin această construcție, elementele tabloului *tab* se inițializează cu valorile:

```
tab[0][0]=-1, tab[0][1]=0, tab[0][2]=1, tab[0][3]=-1;
tab[1][0]=-1, tab[1][1]=0, tab[1][2]=1, tab[1][3]=2;
tab[2][0]=10, tab[2][1]=20, tab[2][2]=30, tab[2][3]=40.
```

2.

```
int tab[][4]={
    {-1,0,1,-1},
    {-1,0,1,2},
    {10,20,30,40}
};
```

Această construcție este identică, ca efect, cu cea din exemplul precedent.

3.

```
double t[][3]={
    {0,1},
    {-1},
    {1,2,3}
};
```

*t* reprezintă o matrice de ordinul 3\*3 ale cărei elemente se inițializează astfel:

```
t[0][0]=0, t[0][1]=1;
t[1][0]=-1;
t[2][0]=1, t[2][1]=2, t[2][2]=3.
```

Elementele *t*[0][2], *t*[1][1] și *t*[1][2] nu sînt inițializate. Ele au valoarea inițială zero dacă tabloul este global sau valori inițiale nedefinite dacă tabloul este automatic.

Construcțiile utilizate pentru inițializarea tablourilor bidimensionale se extind imediat pentru tablouri cu un număr mai mare de dimensiuni.

### Exemplu:

```
int t3[][3][2]={
    {{100,200},{-1,1},{0,1}},
    {{123,456},{789,10},{11,12}}
};
```

### Exerciții:

- 6.2 Să se scrie un program care citește un întreg pozitiv ce reprezintă o sumă exprimată în lei. Se cere să se afișeze numărul minim de bancnote și monede de 10000 lei, 5000 lei, 1000 lei, 500 lei, 200 lei etc. necesare pentru a exprima suma respectivă.

Procesul de calcul a fost indicat în exercițiul 4.21. În cazul de față nu se mai folosesc instrucțiuni de atribuire pentru a da valori inițiale elementelor tabloului *mon* ci ele se inițializează prin declarația tabloului *mon*.

### PROGRAMUL BV12

```
#include <stdio.h>
#include <stdlib.h>

main() /* exprima o suma de lei in bancnote si monede de
        10000 lei, 5000 lei, 1000 lei etc. */
{
    int
    mon[]={1,5,10,20,50,100,200,500,1000,5000,10000};
    int i=sizeof mon/sizeof(int) -1;
    long s,q;
    char t[255];

    for ( ; ; ) {
        printf("suma= ");
        if(gets(t) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%ld",&s) == 1 && s > 0 )
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
    } /* sfirsit for */
    do {
        q=s/mon[i];
```

```

if(q)
    if(i<6) /* monede */
        if( q == 1 ) /* o moneda */
            if(i==0) /* o moneda de 1 leu */
                printf("o moneda de 1 leu\n");
            else
                printf("o moneda a %d lei\n",
                    mon[i]);
        else /* mai multe monede */
            if(i==0) /* monede de 1 leu */
                printf("%ld monede de 1 leu\n",
                    q);
            else /* monede a mon[i] lei */
                printf("%ld monede a %d lei\n",
                    q,mon[i]);
    else /* bancnote */
        if(q==1) /* o bancnota */
            printf("o bancnota a %d lei\n",
                mon[i]);
        else /* mai multe bancnote */
            printf("%ld bancnote a %d lei\n",
                q,mon[i]);
} while ( s != mon[i--]);
}

```

#### Observație:

La declararea lui *i* s-a făcut inițializare cu ajutorul expresiei:

```
sizeof mon / sizeof(int) - 1
```

Această expresie are ca valoarea numărul elementelor tabloului *mon* micșorat cu 1, adică indicele ultimului element al tabloului. Într-adevăr, *sizeof mon* are ca valoare dimensiunea, în octeți, a zonei de memorie alocată tabloului *mon*, iar *sizeof(int)* are ca valoare numărul octeților necesari pentru a reprezenta o dată de tip *int*. Raportul lor ne dă chiar numărul elementelor tabloului *mon*.

#### 6.3 Să se scrie o funcție care afișează elementele unui tablou de tip *char* și citește un întreg de tip *int*.

Funcția are parametrul *text* care este un tablou unidimensional de tip *char*. Ea afișează șirul de caractere păstrat în tabloul *text* înainte de a

citi întregul. Întregul citit se atribuie variabilei globale *v\_int*. Funcția returnează valoarea zero la întâlnirea sfârșitului de fișier și 1 în caz contrar. La eroare, se reia citirea întregului după retastarea acestuia de către operator.

## FUNCȚIA BV13

```
int cit_int(char text[])
/* - citește un întreg și-l atribuie variabilei globale v_int;
   - returnează:
       0 - la întâlnirea sfârșitului de fișier;
       1 - altfel. */
{
    char t[255];
    char texter[]="nu s-a tastat un întreg\n";
    extern int v_int;

    for ( ; ; ) {
        printf(text);
        if(gets(t) == NULL)
            return 0;
        if(sscanf(t,"%d",&v_int) == 1)
            return 1;
        printf(texter);
    }
}
```

6.4 Să se scrie o funcție care citește un întreg de tip *int* care aparține unui interval dat.

Funcția are parametri:

- text* - Tablou unidimensional de tip *char* care are aceeași semnificație ca în exercițiul precedent.
- inf* - Întreg de tip *int* care reprezintă limita inferioară a intervalului căreia trebuie să-i aparțină numărul citit.
- sup* - Întreg de tip *int* care reprezintă limita superioară a aceluiași interval.

Funcția atribuie întregul citit variabilei globale *v\_int*. Ea returnează valoarea zero la întâlnirea sfârșitului de fișier și 1 în caz contrar.

## FUNCȚIA BV14

```
int cit_int(char []); /* prototip */
int cit_int_lim(char text[], int inf, int sup)
/* - citește un întreg de tip int ce aparține intervalului [inf,sup] și-l
   atribuie variabilci v_int;
   - returnează:
       0 - la întâlnirea sfîrșitului de fisier;
       1 - altfel. */
{
    extern int v_int;

    for( ; ; ) {
        if(cit_int(text) == 0)
            return 0; /* s-a întâlnit EOF */
        if(v_int >= inf && v_int <= sup)
            return 1;
        printf("întregul tastat nu aparține\
              intervalului:");
        printf("[%d,%d]\n", inf,sup);
        printf("se reia citirea\n");
    }
}
```

6.5 Să se scrie o funcție care validează o dată calendaristică.

Funcția are 3 parametri de tip *int*:

<i>zi</i>	- Numărul zilei din cadrul lunii.
<i>luna</i>	- Numărul lunii.
<i>an</i>	- Anul calendaristic.

Funcția returnează:

<i>1</i>	- Dacă data calendaristică este validă.
<i>0</i>	- Altfel.

În exercițiile din această carte se presupune că anul calendaristic aparține intervalului [1600,4900]. În acest caz, anul este bisect dacă expresia:

$$an \% 4 == 0 \ \&\& \ an \% 100 != 0 \ || \ an \% 400 == 0$$

are valoarea adevărat (vezi exercițiul 3.12.).

Funcția utilizează un tablou global de tip *int* ale cărui elemente au ca valori numărul zilelor din lunile calendaristice. Numele acestui tablou este *nrzile*.

nrzile[i] are ca valoare numărul zilelor din luna a i-a ( $i=1,2,\dots,12$ ).

## FUNCȚIA BVIS

```
int v_calend(int zi,int luna,int an)
/*- valideaza data calendaristica;
   - returneaza:
       1 - daca este corecta,
       0 - altfel. */
{
    extern int nrzile[];

    if(an < 1600 || an > 4900 ) {
        printf("anul nu este in intervalul\
               [1600,4900]");
        return 0;
    }
    if(luna < 1 || luna > 12 ) {
        printf("luna=%d eronata\n", luna);
        return 0;
    }
    if(zi < 1 || zi > nrzile[luna] +
        (luna == 2 && ( an%4 == 0 &&
            an%100 || an%400 == 0 )))
    {
        printf("ziua=%d eronata\n",zi);
        return 0;
    }
    return 1; /* data calendaristica valida */
}
```

### Observație:

Elementul *nrzile[2]* are ca valoare numărul zilelor din luna februarie pentru un an nebisect, deci 28.

Dacă anul este bisect, atunci expresia:

$$(1) \text{ an}\%4=0 \ \&\& \ \text{an}\%100 \neq 0 \ \vee \ \text{an}\%400=0$$

are valoarea 1. În acest caz numărul zilelor din luna februarie se mărește cu 1. Aceasta nu este valabil și pentru celelalte luni. Deci

*nrzile[luna]*

se mărește cu 1 cind expresia (1) are valoarea 1 și cind expresia

$(2) \text{ luna} = 2$

este adevărată. Cu alte cuvinte, numărul zilelor din februarie se mărește cu 1, cînd atît expresie (1), cit și expresie (2) sînt adevărate și numai atunci. Aceasta înseamnă că expresia:

(2) && (1)

are valoarea 1 atunci și numai atunci cînd anul este bisect și luna este februarie.

6.6 Să se scrie o funcție care determină dintr-o dată calendaristică de forma zi, lună și an, numărul zilei din an pentru data respectivă.

Funcția returnează numărul determinat din parametri *zi*, *luna* și *an*.

Dacă, de exemplu, data calendaristică este 1 martie 1994, atunci ea reprezintă ziua 60 din anul respectiv. Într-adevăr, pînă la 1 martie 1994 au trecut lunile ianuarie și februarie, deci împreună cu 1 martie sînt:

$31 + 28 + 1 = 60$  zile

Într-un an bisect data de 1 martie este ziua 61 din anul respectiv.

Procesul de calcul constă din însumarea zilelor din lunile precedente lunii calendaristice definită de *luna*, la ziua curentă definită de *zi*.

Funcția de față presupune că data calendaristică este validă.

## FUNCȚIA BV16

```
int zi_din_an(int zi, int luna, int an)
/* determina ziua din an si returneaza numarul astfel determinat */
{
    extern int nrzile[];
    int bisect = an%4 == 0 && an%100 || an%400 == 0;
    int i;

    for( i=1; i < luna; i++ )
        zi += nrzile[i] + (i == 2 && bisect);
    return zi;
}
```

6.7 Să se scrie o funcție care dintr-o dată calendaristică definită prin numărul zilei din an și anul respectiv, determină luna și ziua din luna respectivă.

Această funcție este inversa funcției *zi\_din\_an* definită în exercițiul

precedent. Ea definește două valori:

numărul lunii

și

numărul zilei din luna respectivă.

Funcția poate fi realizată în una din variantele:

- Să existe un parametru formal care să fie un tablou de tip *int*. Fie acesta *tzll* și care să corespundă unui parametru efectiv care să fie un tablou de 2 elemente de tip *int*. În acest caz se poate atribui, de exemplu, lui *tzll[0]* numărul zilei din lună și lui *tzll[1]* numărul lunii.
- Să existe două variabile globale la care să li se atribuie cele două valori: ziua din lună și respectiv luna.

Funcția realizată mai jos utilizează parametrul *tzll*

## FUNCȚIA BVI7

```
void luna_si_ziua ( int zz, int an,int tzzll[] )
/* - determina luna si ziua din luna;
   zz - ziua din an;
   an - anul calendaristic;
   - la revenire:
      tzzll[0] are ca valoare ziua din luna;
      tzzll[1] are ca valoare luna calendaristica */
{
  int bisect=an%4==0 && an%100 || an%400 ==0;
  int i;
  extern int nrzile[];

  for(i=1; zz > nrzile[i] + (i==2 && bisect);i++)
    zz -= (nrzile[i] + ( i==2 && bisect));
  tzzll[0] = zz;
  tzzll[1] = i;
}
```

6.8 Să se scrie o funcție care citește o dată calendaristică compusă din zi, luna și an.

Funcția validează data calendaristică respectivă. Ea returnează valoarea:

- 0 - La întâlnirea sfârșitului de fișier.
- 1 - Altfel (data calendaristică corectă).



Are un parametru *tzzllaa*, care este un tablou de tip *int*. La revenire, elementele acestui tablou au valorile:

*tzzllaa*[0]        - Ziua citită.  
*tzzllaa*[1]        - Luna citită.  
*tzzllaa*[2]        - Anul citit.

## FUNCȚIA BV18

```
int cit_data_calend( int tzzllaa[] )
/* - citește o data calendaristică, o validează și o păstrează în
   tabloul tzzllaa:
       tzzllaa[0] - ziua;
       tzzllaa[1] - luna;
       tzzllaa[2] - anul;
   - funcția returnează:
       0 - la întâlnirea sfârșitului de fișier;
       1 - altfel */
{
    extern int v_int;
    static char ziua[] = "ziua: ";
    static char luna[] = "luna: ";
    static char anul[] = "anul: ";
    static char eroare[] = "s-a tastat EOF";

    for( ; ; ) {

        /* se citește ziua */
        if( cit_int_lim( ziua, 1, 31 ) == 0 ) {
            printf( "%s\n", eroare );
            return 0;
        }
        tzzllaa[0] = v_int;

        /* se citește luna */
        if( cit_int_lim( luna, 1, 12 ) == 0 ) {
            printf( "%s\n", eroare );
            return 0;
        }
        tzzllaa[1] = v_int;

        /* se citește anul */
        if( cit_int_lim( anul, 1600, 4900 ) == 0 ) {
```

```

        printf("%s\n", eroare);
        return 0;
    }
    tzzllaa[2] = v_int;

    /* validare data calendaristica */
    if(v_calend(tzzllaa[0], tzzllaa[1], tzzllaa[2]))
        return 1;
    printf("data calendaristica este eronata\n");
    printf("se reia citirea datei\
        calendaristice\n");
}
}

```

- 6.9 Să se scrie un program care citește o dată calendaristică și afișează data calendaristică pentru ziua următoare.

Programul se realizează conform următorilor pași:

1. Se citește și se validează data calendaristică prin apelul funcției *v\_calend*.
2. Dacă data citită este 31 decembrie dintr-un anumit an, atunci ziua următoare este 1 ianuarie din anul următor, apoi se trece la pasul 6.  
Altfel se trece la pasul 3.
3. Se determină ziua din an (funcția *zi\_din\_an*).
4. Se incrementează ziua din an.
5. Se determină luna și ziua din lună (funcția *luna\_si\_ziua*).
6. Se afișează data calendaristică a zilei următoare.

## PROGRAMUL BV19

```

#include <stdio.h>
#include <stdlib.h>
#include "bvi3.cpp" /* funcția cit_int */
#include "bvi4.cpp" /* funcția cit_int_lim */
#include "bvi5.cpp" /* funcția v_calend */
#include "bvi6.cpp" /* funcția zi_din_an */
#include "bvi7.cpp" /* funcția luna_si_ziua */
#include "bvi8.cpp" /* funcția cit_data_calend */

```

```

/* variabile globale */

```

```

int v_int; /* se utilizeaza in functiile:
           cit_int;
           cit_int_lim;
           cit_data_calend. */

int nrzile[]={0,31,28,31,30,31,30,31,31,30,
              31,30,31};
/* se utilizeaza in functiile:
   v_calend;
   zi_din_an;
   luna_si_ziua */

main() /* citeste o data calendaristica, o valideaza si in caz ca este
       corecta, afiseaza data calendaristica a zilei urmatoare */
{
    int data_calend[3];
    int zl[2];

    /* citeste si valideaza data calendaristica */
    if(cit_data_calend(data_calend) == 0)
        exit(1);
    if(data_calend[0] == 31 && data_calend[1] == 12 )
    {
        /* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
        zl[0] = 1; /* 1 */
        zl[1] = 1; /* ianuarie */
        data_calend[2]++; /* anul urmator */
    }
    else /* se determina ziua urmatoare */
        luna_si_ziua(zi_din_an(data_calend[0],
                                data_calend[1],data_calend[2])+1,
                     data_calend[2],zl);

    /* afiseaza data calendaristica a zilei urmatoare */
    printf("ziua:%d\tluna:%d\tanul:%d\n",
           zl[0],zl[1],data_calend[2]);
}

```

## 7. PROGRAMARE MODULARĂ

Un program poate fi format din mai multe *module*.

Numim *modul sursă*, o parte a textului sursă al programului care se compilează printr-o singură compilare și separat de restul textului sursă al programului respectiv. Rezultatul compilării unui modul sursă este un *modul obiect*.

În cele ce urmează, prin *modul* vom înțelege un modul sursă. Modulele obiect le vom mai numi și module de tip *OBJ*.

De obicei, programele simple se compun dintr-un singur modul sau un număr relativ mic de module (4-5).

În componența unui modul intră proceduri "înrudite". Această noțiune nu este definită riguros. De obicei, spunem ca mai multe proceduri sînt înrudite dacă utilizează în comun diferite date. De exemplu, funcțiile *zi din an* și *luna și ziua*, definite în capitolul precedent, se consideră că sînt înrudite. Ele utilizează în comun tabloul global *nrzile*.

Un modul se compune din unul sau mai multe fișiere. În cazul în care el se compune din mai multe fișiere, acestea se compilează împreună incluzîndu-le unul în altul cu ajutorul construcției *#include*.

Modulele de tip *OBJ*, obținute prin compilările modulelor din compunerea unui program, pot fi reunite într-un fișier executabil (cu extensia *.EXE*) cu ajutorul editorului de legături.

Modulele unui program se definesc ca rezultat al procesului de *descompunere* a unei probleme în *subprobleme mai simple*. Acest proces de descompunere se poate continua, adică subproblemele obținute la o primă descompunere pot fi și ele la rîndul lor descompuse în altele mai simple și așa mai departe, pînă cînd se ajunge ca toate subproblemele de la nivelele inferioare să fie relativ simple. Diferite componente ale unei astfel de descompuneri se realizează prin module. De obicei, aceste module se pot realiza în paralel și relativ independent de către mai mulți programatori, ceea ce conduce la creșterea eficienței în programare.

Un avantaj mare pe care îl oferă modulele, este așa numita posibilitate de "ascundere a datelor". Aceasta înseamnă că la o parte sau chiar la toate datele unui modul au acces direct numai procedurile din compunerea modulului respectiv. Adesea se spune că datele respective sînt *vizibile* numai la nivel de modul. Ele pot fi utilizate în alte module numai prin intermediul procedurilor modulului în care sînt vizibile. Mai mult decît atît, pot exista cazuri în care este avantajos ca și anumite proceduri dintr-un modul să fie

*ascunse* La acestea nu se pot face apeluri din alte module. Ele pot fi apelate numai de procedurile modulului în care sînt definite și ascunse

Facilitatea de "ascundere" a datelor și procedurilor în module constituie o *protecție* împotriva deteriorării datelor prin accese neautorizate din alte module. Această protecție devine importantă mai ales în cazul problemelor complexe, cînd participă colective mari pentru programarea și implementarea lor.

Prin *programarea modulară* înțelegem stilul de programare care are la bază utilizarea de module în vederea "ascunderii" datelor și procedurilor pentru a realiza protecția datelor respective față de accese neautorizate.

Limbajul C a fost proiectat în ideea de a permite utilizarea programării modulare. În acest caz modulul conține funcții "înrudite". Datele statice, declarate în afara funcțiilor modulului, pot fi utilizate în comun de către aceste funcții. Ele sînt ascunse în modul, deoarece funcții din alte module nu pot face acces direct la ele.

De asemenea, se pot defini și funcții statice, ceea ce conduce la ascunderea lor în modulul în care au fost definite.

O funcție statică poate fi apelată numai de funcții definite în același modul cu ea.

### Exerciții:

7.1 Să se realizeze un modul care implementează o stivă ale cărei elemente sînt de tip *int*.

Prin *stivă* înțelegem o mulțime ordonată de elemente la care se are acces în conformitate cu principiul *LIFO* (*Last In First Out*)

Cel mai simplu procedeu de implementare a unei stive este păstrarea elementelor ei într-un tablou unidimensional. Ulterior vom vedea și alte moduri de implementare a unei stive.

În zona de memorie alocată stivei se pot păstra elementele ei, unul după altul. De asemenea, ele se pot scoate din zona de memorie respectivă, unul câte unul, în ordine inversă păstrării lor.

Ultimul element pus pe stivă se spune că este în *virful* stivei. Primul element pus pe stivă se află la *baza* stivei.

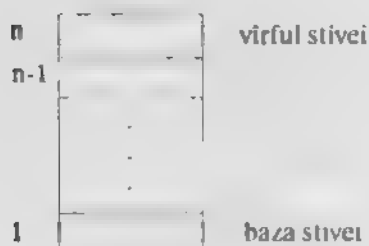
Virful stivei se modifică atunci cînd se pune un element pe stivă sau cînd se scoate de pe stivă. În primul caz *lungimea* stivei (numărul elementelor din stivă) crește, iar în cel de al doilea caz descrește.

Într-adevăr, cînd se pune un element pe stivă, atunci acesta se pune după cel aflat în virful stivei și prin aceasta virful stivei se modifică așa încît

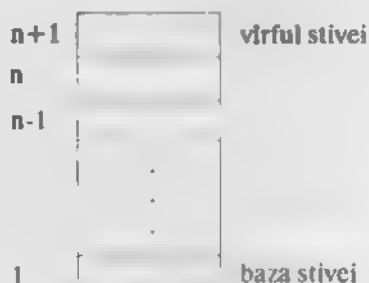
elementul nou pus pe stivă să fie în vârful ei. Când se ia un element de pe stivă, atunci acesta este cel aflat în vârful stivei și apoi vârful stivei se modifică așa încît, elementul care a fost pus pe stivă înaintea celui scos, să fie în vârful ei.

### Punerea unui element pe stivă

*Înainte de a pune  
un element pe stivă*



*După ce s-a pus  
un element pe stivă*

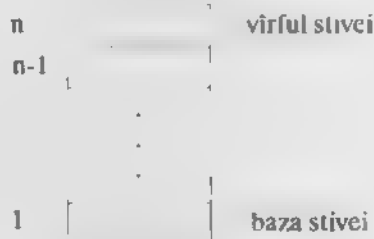


### Scoaterea unui element de pe stivă

*Înainte de a scoate  
un element de pe stivă*



*După ce s-a scos  
un element de pe stivă*



Se observă că totdeauna, ultimul element pus pe stivă este primul care se scoate din stivă. De aici provine afirmația că o stivă se gestionează conform principiului *LIFO*.

Pentru a implementa o stivă vor fi necesare două funcții:

- una care să pună un element pe stivă

și

- una care să scoată un element din stivă.

Aceste funcții au denumiri deja consacrate în literatura de specialitate și anume, funcția care pune un element pe stivă se numește *push*, iar cealaltă *pop*.

Vom păstra și noi aceste denumiri.

Am spus mai sus că vom implementa stiva folosind un tablou unidimensional. Acesta va fi în cazul de față de tip *int*. Numim *stack* acest tablou.

`stack[0]` este elementul de la baza stivei.

Funcțiile *push* și *pop* gestionează vârful stivei și de aceea ele au nevoie de o variabilă care să definească indicele elementului din vârful stivei. De obicei, se păstrează indicele *primului element liber*, adică indicele elementului tabloului *stack* care urmează imediat după elementul aflat în vârful stivei (vezi [2]).

Numim *next* variabila a cărei valoare curentă este primul element liber al tabloului *stack*.

Evident inițial *next*=0, deoarece la început nefiind nici un element pe stivă, `stack[0]` este primul element liber.

Utilizatorul stivei nu trebuie să aibă acces direct la elementele stivei și de aceea atât tabloul *stack*, cât și variabila *next* trebuie "ascunse" în modul. În felul acesta, utilizatorul are posibilitatea să pună un element pe stivă în vârful ei apelând funcția *push* și să scoată elementul din vârful stivei apelând funcția *pop*. Rezultă că aceste două funcții nu se "ascund" în modul

De obicei, se definesc și alte funcții (vezi [5]) cum ar fi:

- |              |  |
|--------------|--|
| <i>clear</i> | - Pentru a vida stiva.   |
| <i>empty</i> | - Stabilește dacă stiva este vidă sau nu.                                  |
| <i>full</i>  | - Stabilește dacă stiva este plină sau nu.                                 |
| <i>top</i>   | - Permite acces la elementul din vârful stivei, fără a-l scoate din stivă. |

Toate aceste funcții trebuie să aibă acces atât la *stack*, cât și la variabila *next*.

De aceea, *stack* și *next* se vor declara statice în afara corpurilor lor.

Cele 6 funcții amintite mai sus vor avea un caracter global, ele putând fi apelate din orice modul al programului.

Funcția *push* are prototipul:

```
void push(int x);
```

În principiu, ea pune pe stivă valoarea lui *x*, deci trebuie să facă atribuirea.

```
stack[next]=x
```

și apoi să incrementeze pe *next* pentru ca aceasta să definească locul liber curent. De aceea, atribuirea de mai sus se poate realiza împreună cu incrementarea lui *next*:

```
stack[next++] = x.
```

Funcția trebuie să testeze, în prealabil, dacă există loc liber pentru a păstra pe *x* și numai în acest caz se va executa atribuirea de mai sus. În caz că stiva este plină (depășită), se dă un mesaj de eroare.

Funcția *pop* are prototipul:

```
int pop(void);
```

Ea returnează valoarea din vârful stivei. În acest scop se va decrementa *next* și apoi se va returna valoarea elementului

```
stack[next].
```

Aceste două acțiuni se pot realiza cu ajutorul instrucțiunii

```
return stack[--next];
```

Se observă că prin aceasta vârful stivei coboară cu un element, deoarece *next*, care exprimă primul element liber, este chiar indicele elementului eliminat de pe stivă.

Înainte de a executa instrucțiunea de mai sus, funcția *pop* va testa dacă stiva nu cumva este vidă. În cazul în care stiva este vidă, funcția *pop* va afișa un mesaj de eroare și va returna valoarea zero.

Funcția *top* este ca și *pop*, cu deosebirea că nu se elimină elementul din vârful stivei ci numai se returnează valoarea lui.

Funcția *clear* are prototipul:

```
void clear(void);
```

Ea videază stiva, ceea ce se realizează simplu prin atribuirea valorii zero variabilei *next*. În felul acesta, tot tabloul devine liber.



Funcția *empty* are prototipul:

```
int empty(void);
```

Ea returnează valoarea 1 dacă stiva este vidă și zero în caz contrar.

Funcția *full* are prototipul:

```
int full(void);
```

Ea returnează valoarea 1 dacă stiva este plină și zero în caz contrar.

Aceste funcții, împreună cu declarațiile lui *stack* și *next* se păstrează într-un fișier care formează modulul ce implementează stiva *stack* prin intermediul unui tablou de tip *int*.

## MODULUL BVIII1

```
#define MAX 1000
```

```
static int stack[MAX];
```

```
static int next = 0;
```

```
void push( int x) /* pune x pe stiva */
```

```
{
```

```
    if(next < MAX )
```

```
        stack[next++] = x;
```

```
    else
```

```
        printf("stiva este plina\n");
```

```
}
```

```
int pop() /* scoate din stiva elementul din virful ei */
```

```
{
```

```
    if( next > 0 )
```

```
        return stack[--next];
```

```
    else
```

```
        printf("stiva vida\n");
```

```
}
```

```
int top() /* returneaza elementul din virful stivei */
```

```
{
```

```
    if(next > 0 )
```

```
        return stack[next-1];
```

```
    else
```

```
        printf("stiva vida\n");
```

```
}
```

```

void clear() /* videaza stiva */
{
    next = 0;
}

int empty() /* returneaza 1 daca stiva este vida si 0 altfel */
{
    return !next;
}

int full() /* returneaza 1 daca stiva este plina si 0 altfel */
{
    return next == MAX;
}

```

7.2 Să se scrie un program care transcrie o expresie cu operanzi numere naturale, în formă poloneză postfixată.

Prin expresie aritmetică înțelegem o expresie în care pot fi utilizate numai cele 4 operații binare: adunare (+), scădere (-), înmulțire (\*) și împărțire (/).

Se pot utiliza parantezele rotunde pentru a schimba prioritatea operatorilor.

Forma *poloneză postfixată* a unei expresii aritmetice se poate defini ca mai jos:

- forma poloneză postfixată a unui operand coincide cu el însuși dacă nu este inclus în paranteze rotunde;
- forma poloneză postfixată a unui operand de forma  
(exp)

coincide cu forma poloneză postfixată a lui *exp*;

- dacă operatorul binar *op* leagă expresiile *fexp1* și *fexp2* care sînt în formă poloneză postfixată:

*fexp1 op fexp2*

atunci forma poloneză postfixată a acestei expresii este

*fexp1 fexp2 op*

### Exemple:

Expresie aritmetică	Echivalentul ei în formă poloneză postfixată
$1+2$	$1\ 2\ +$
$1+2*3$	$1\ 2\ 3\ *\ +$
$(1+2)*3$	$1\ 2\ +\ 3\ *$

Un algoritm simplu pentru a transforma o expresie aritmetică în formă poloneză postfixată se bazează pe utilizarea unei stive în care se păstrează temporar operatorii expresiei. În acest scop se va folosi stiva implementată prin modulul din exemplul precedent.

Expresia aritmetică în formă poloneză postfixată, pe măsură ce se construiește, se păstrează într-un tablou *tefpp* de tip caracter.

Algoritmul de traducere are următorii pași:

- Un operand se păstrează automat în tabloul *tefpp*. Acest lucru rezultă din faptul că prin această transformare operanzii își păstrează ordinea din expresia inițială. Apoi se continuă cu următorul element al expresiei.
- Paranteza deschisă se introduce automat în stivă.
- Un operator se introduce în stivă dacă în vârful stivei se află:
  - paranteza deschisă;
  - un operator de prioritate mai mică;sau
  - stiva este vidă.

Dacă în vârful stivei se află un operator de prioritate mai mare sau egală cu a celui curent, atunci operatorul din vârful stivei se scoate din stivă și se păstrează în tabloul *tefpp*. Apoi se compară din nou operatorul din vârful stivei cu cel curent și se reia pasul c.

- La întâlnirea unei paranteze închise, se scot pe rând operatorii din stivă și se introduc în tabloul *tefpp*, până la întâlnirea parantezei deschise din stivă. Aceasta pur și simplu se elimină din stivă. Apoi se continuă cu următorul element al expresiei.
- La terminarea parcurgerii expresiei, se scot pe rând toți operatorii care se mai află în stivă și se trec în *tefpp*.

Se presupune că expresia se termină cu punct și virgulă.

La baza stivei se va afla caracterul NUL ("0").

Programul de față poate transforma mai multe expresii, până la tastarea sfârșitului de fișier.

Menționăm că metoda de transcriere a expresiilor descrisă mai sus, funcționează exact numai dacă expresia respectivă este corectă din punct de vedere sintactic.

În mod normal acest program se compune din două module, unul care a fost definit în exercițiul precedent (fișierul sursă BVII1.CPP) și prin care se implementează o stivă pe care se pun elementele de tip *int*, iar cel de al doilea modul definește funcția principală.

Cele două module se pot compila distinct obținându-se două module de tip OBJ care urmează apoi a fi link-editate împreună pentru a obține imaginea executabilă a programului.

Cele două fișiere de tip sursă pot fi compilate și împreună, procedând ca până acum, adică incluzând fișierul BVII1.CPP în fișierul BVII2.CPP.

De asemenea, fișierele respective pot fi compilate și link- editate folosind un fișier de tip *Project*. Acesta se editează utilizând meniul *Project* al mediului integrat de dezvoltare Turbo C++.

Mai jos s-a adoptat varianta cu includerea fișierului BVII1.CPP.

Fișierul BVII2.CPP, alături de funcția principală mai conține o funcție denumită *sca*. Aceasta are prototipul:

```
int sca(int c);
```

Ea realizează saltul peste caracterele albe.

Dacă *c* are ca valoare codul ASCII al unui caracter alb, atunci se citesc caracterele care urmează, până la întâlnirea primului caracter care nu este alb. La revenire, se returnează codul ultimului caracter citit. Dacă *c* are ca valoare codul unui caracter ASCII care nu este alb, atunci se revine din funcție cu codul caracterului respectiv.

**PROGRAMUL BVII2.CPP**

```
#include <stdio.h>
#include "bvii1.cpp"

#define MAXTEFPP 1000

int sca(int); /* prototipul functiei sca */
```

```

/* prototipul functiilor push, pop si clear sint necesare cind cele doua
   fisiere se compileaza separat; in cazul de fata sint in plus */
void push(int);
int pop(void);
void clear(void);

main() /* transforma expresii aritmetice
        in forma poloneza postfixata */
{
    int c,i,j;
    char tefpp[MAXTEFPP];

    for ( c=getchar(); c != EOF; ) {
        i=0; /* indice pentru tefpp */
        clear(); /* videaza stiva */
        push('\0'); /* zero la baza stivei */
        while((c=sca(c)) != ';' && c != EOF ) {

/* 1 */

/* c nu este alb, punct si virgula sau sfirsitul de fisier */
            while(c>='0' && c<='9') { /* 2 */

/* se pastreaza un operand */
                tefpp[i++] = c;
                c = getchar();
            } /* sfirsit while 2 */

/* s-a terminat un operand, poate urma un caracter alb, un operator,
   punct si virgula sau EOF */
            c = sca(c) ; /* salt peste caracterele albe daca exista */
            tefpp[i++] = ' '; /* spatiu dupa operand */
            switch(c) {
                case '(': /* paranteza deschisa se pune pe stiva */
                    push(c);
                    break;
                case ')': /* paranteza inchisa: se scot operatorii
                           din stiva pina la paranteza deschisa
                           si se trec in tefpp */
                    while((c=pop()) != '(')
                        tefpp[i++] = c;
                    break;

```

```

        case '+':
        case '-': /* se scot operatorii de pe stiva si se trec in
                    tabloul tcfpp */
            while((j=pop())=='+' ||
                    j=='-' || j=='*' || j=='/')
                tcfpp[i++] = j;

/* ultimul element scos de pe stiva se repune */
        push(j);

/* se pune pe stiva operatorul curent */
        push(c);
        break;

        case '*':
        case '/': /* se scot de pe stiva operatorii * si / */
            while((j=pop()) != '*' || j != '/')
                tcfpp[i++] = j;

/* ultimul element scos de pe stiva se repune */
        push(j);

/* se pune pe stiva operatorul curent */
        push(c);
        break;

        case ';': /* sfirsit expresie */
            break;

        default:
/* expresie eronata */
            printf("caracter eronat:%c\t%d\n",
                    c,c);

/* se cauta sfirsitul expresiei sau EOF */
            while( c != ';' && c != EOF )
                c=getchar();
        } /* sfirsit switch */
        if( c != ';' && c != EOF )

/* avans la caracterul urmator din expresie */
        c = getchar();
    } /* sfirsit while 1 */

```

```

/* sfirsit expresie: se trec operatorii din stiva in tefpp,
daca exista */
    while((j = pop()) != '\0')
        tefpp[i++] = j;

/* se afiseaza forma poloneza postfixata a expresiei curente */
    putchar('\n');
    for(j=0; j<i; j++)
        putchar(tefpp[j]);
    putchar('\n');

/* la EOF se termina executia; altfel se citeste primul caracter al
expresiei urmatoare */
    if(c == EOF )
        break;
    c = getchar();
    c = sca(c); /* avans peste caractere albe */

} /* sfirsit for neimbricat */
} /* sfirsit main */

int sca(int x) /* salt peste caractere albe */
{
    while(x == ' ' || x == '\t' || x == '\n')
        x = getchar();
    return x;
}

```

## 8. POINTERI

Un *pointer* este o variabilă care are ca valori adrese. Pointerii se utilizează pentru a face referire la date cunoscute prin adresele lor. Astfel dacă  $p$  este o variabilă de tip *pointer* care are ca valoare adresa lui  $x$ , atunci

$*p$

reprezintă chiar valoarea lui  $x$

Fie de exemplu:

$\text{int } x, y;$

atunci dacă  $p$  are ca valoare adresa lui  $x$ , atribuirea:

$y = x + 100$

este identică cu:

$y = *p + 100$

În mod analog, atribuirea:

$x = 3$

este identică cu:

$*p = 3$

În construcția  $*p$  utilizat, mai sus, caracterul  $*$  se consideră că fiind un operator unar care furnizează valoarea din zona de memorie a cărei adresă este conținută în  $p$ .

Operatorul unar  $*$  are aceeași prioritate ca și ceilalți operatori unari din limbajul C și se asociază de la dreapta spre stânga (vezi 5.2.16).

Dacă  $p$  conține adresa zonei de memorie alocată lui  $x$ , vom spune că  $p$  *pointează* spre  $x$ .

De asemenea, dacă  $p$  are ca valoare adresa de început a unei zone de memorie care conține o dată de tipul  $tp$ , atunci vom spune că  $p$  *pointează* spre  $tp$ .

Menționăm că în legătură cu noțiunea de *pointer*, în limba română se utilizează și alte denumiri, ca de exemplu

- referință;
- localizator;
- reper;



- indicator de adresă etc.

În cartea de față păstrăm denumirea englezească.

Pentru a atribui o adresă unei variabile de tip *pointer* se poate folosi operatorul unar & (vezi 3.2.11.). Astfel, dacă dorim ca *p* să poarte spre *x* (să aibă ca valoare adresa lui *x*), atunci putem utiliza atribuirea:

$p = \&x$

Operatorul unar & este numit operator *adresă* sau de *referențiere*. Operatorul unar \* îl vom numi operator de *indirectare* sau de *dereferențiere*.

Ultima denumire a operatorului unar \* decurge din efectul invers al acestuia față de operatorul unar &. Într-adevăr, expresia:

• **SLX**

are aceeași valoare ca și operandul  $x$ .

**Exemplu:**

**Fie declarația:**

```
int x;
```

Variabilei  $x_1$  se alocă o zonă de memorie de 16 biți:



File 1000 adresa zonei alocate variabilei x.

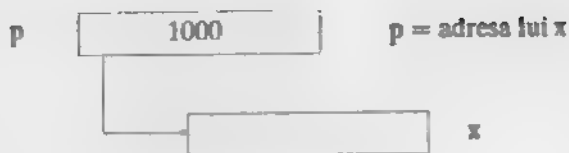
Dacă  $p$  este o variabilă pointer, atunci ei i se atribuie, de asemenea, o zonă de memorie de 16 biți (o să vedem ulterior că uneori această zonă va fi de 32 biți):



Folosind instrucțiunea de atribuire:

 $p = \alpha;$ 

lui  $p$  i se atribuie valoarea 1000:



Instrucțiunea:

```
x = 10;
```

atribuie lui `x` valoarea 10:



Același efect se obține folosind instrucțiunea:

```
*p = 10;
```

În concluzie, instrucțiunea:

```
x = 10;
```

este echivalentă cu secvența:

```
p = &x;
*p = x;
```

sau

```
p = &x, *p = x;
```

Noțiunea de pointer joacă un rol important deoarece permite calcule cu adrese. Acestea sînt utile mai ales în scrierea programelor de sistem. Astfel de calcule sînt proprii limbajelor de asamblare. Introducerea lor în limbajul C, oferă programatorilor posibilitatea de a scrie programe mai optime, decît cele realizate fără a beneficia de facilitățile oferite de pointeri.

Un beneficiu obținut simplu pe baza utilizării pointerilor este înlocuirea expreselor cu indici prin expresii cu pointeri. Ca efect, înmulțirile utilizate la evaluarea variabilelor cu indici se schimbă cu adunări și deplasări. Un alt beneficiu obținut cu ajutorul pointerilor este posibilitatea alocării dinamice a memoriei prin alte mijloace decît cel utilizat la alocarea datelor automate.

Folosirea funcțiilor ca parametri este și el un beneficiu al utilizării pointerilor.

În capitolul de față se descriu aspectele de bază cu privire la utilizarea pointerilor în scrierea programelor C.

## 8.1. Declarația de pointer și tipul pointer

Un *pointer* se declară ca orice variabilă, cu singura deosebire că numele este precedat de caracterul \*. Astfel, dacă dorim să declarăm variabila *p* utilizată mai sus pentru a păstra adresa lui *x*, vom folosi declarația:

```
int *p;
```

Tipul *int* stabilește faptul că *p* conține adrese de zone de memorie în care se păstrează date de tip *int*. Declarația de mai sus se poate interpreta astfel:

\**p* reprezintă conținutul zonei de memorie spre care pointează *p*, iar acest conținut are tipul *int*.

În general, un pointer se declară prin:

```
tip *nume;
```

ceea ce înseamnă că *nume* este un pointer care pointează spre o zonă de memorie ce conține o dată de tipul *tip*.

Comparînd declarația de mai sus cu cea obișnuită:

```
tip nume;
```

putem considera că:

```
tip *
```

dintr-o declarație de pointeri reprezintă

```
tip
```

dintr-o declarație obișnuită. De aceea, construcția:

```
tip *
```

se spune că reprezintă un tip nou, tipul *pointer*. Acest tip se spune că este tipul *pointer* spre *tip*.

Dacă avem declarațiile:

```
int x;  
int *p;  
float y;
```

atunci atribuirea

```
p = &x
```

este corectă, în timp ce

```
p = &y
```

nu este corectă, deoarece *p* poate conține numai adrese de zone de memorie în care se păstrează date de tip *int*.

Dacă există declarația:

```
float *q;
```

atunci se poate folosi atribuirea

```
q = &y
```

**Exemple:**

```
int x,y;
```

```
int *p;
```

1.

```
y = x+100;
```

este echivalentă cu secvența:

```
p = &x;
```

```
y = *p+100;
```

2.

```
x = y;
```

este echivalentă cu secvența:

```
p = &x;
```

```
*p = y;
```

3.

```
x++;
```

este echivalentă cu secvența:

```
p = &x;
```

```
(*p)++;
```

Există cazuri în care dorim ca un pointer să fie utilizat cu mai multe tipuri de date. În acest caz, la declararea lui nu putem specifica un tip, ca în exemplele de mai sus. Aceasta se realizează folosind cuvântul *void*:

```
void *nume;
```

**Exemplu:**

```
int x;
```

```

float y;
char c;
void *p;
..
p = &x;
..
p = &y;
..
p = &c;
...

```

Deoarece *p* a fost declarat cu ajutorul cuvintului cheie *void*, lui *p* i se pot atribui adrese de zone de memorie care pot conține date de tipuri diferite.

*int, float, char* etc.

Cînd se folosesc pointeri de tip *void*, este necesar să se facă conversii explicite prin expresii de tip *cast*, pentru a preciza tipul datei spre care pointează un astfel de pointer. Într-adevăr, dacă se utilizează *p* declarat ca mai sus, atunci o atribuire de forma:

```
"p = 10
```

nu este corectă, deoarece nu este definit tipul datei spre care pointează *p*.

Pentru a putea realiza o astfel de atribuire, va trebui să convertim valoarea lui *p* spre tipul "pointer spre tipul *int*". Tipul *pointer spre int* se exprimă prin construcția:

```
int *
```

Amintim că valoarea unui operand se poate converti spre tipul *tip* folosind operatorul unar (*tip*):

(*tip*) *operand*

De exemplu, pentru a converti valoarea lui *y*, din exemplul de mai sus, spre *long*, vom scrie:

```
(long) y
```

În cazul de față, valoarea lui *p* trebuie convertită spre tipul *int \** deci vom folosi expresia cast:

```
(int *) p
```

În felul acesta atribuirea de mai sus devine:

```
*(int *)p = 10
```

În mod analog, o expresie de forma:

**\*p+1.5**

este eronată. Este necesar să se convertească explicit valoarea lui *p* spre tipul de dată conținut în zona de memorie a cărei adresă este valoarea lui *p*. Rezultă că expresiile:

**\*(int \*)p+1.5**

**\*(float \*)p+1.5**

**\*(char \*)p+1.5**

pot fi utilizate în locul expresiei de mai sus. Expresia:

**\*(int \*)p+1.5**

se evaluează astfel:

a. Adresa care este valoarea lui *p* se interpretează ca fiind adresa zonei de memorie care conține o dată de tip *int*.

b. Valoarea de la adresa definită de expresia:

**(int \*)p**

se convertește din *int* spre *double* în conformitate cu regula conversiilor implicite.

c. Valoarea obținută la punctul b se adună cu 1.5 și suma este de tip *double*.

**Conversia tipului pointer:**

**void \***

spre un tip pointer concret (*int \**, *float \** etc.) este totdeauna posibilă și nu înseamnă altceva decât *precizarea* tipului de pointer pe care îl are valoarea pointerului la care se aplică conversia respectivă. Deci, conversia realizată prin expresia *cast*:

**(int \*)p**

precizează faptul că *p* are ca valoare o adresă a unei zone de memorie în care se păstrează întregi de tip *int*.

**Utilizarea tipului:**

**void \***

asigură o flexibilitate mare în utilizarea pointerilor. Cu toate acestea, se recomandă să nu se utilizeze în mod abuziv, deoarece aceasta poate fi și o sursă de erori. Într-adevăr, programatorul trebuie să știe, în fiecare moment, ce fel de tip de pointer este valoarea atribuită variabilei pointer de tip *void \**.

De exemplu, dacă se utilizează expresia:

(int \*)p

Într-un moment în care *p* are ca valoare adresa unei zone de memorie care conține o dată flotantă, rezultatul va fi imprevizibil.

## 8.2. Realizarea apelului prin referință utilizând parametri de tip pointer

Am văzut că în limbajul C apelul este prin valoare. Aceasta înseamnă că la un apel, parametrilor formali li se atribuie valorile parametrilor efectivi care le corespund (vezi 4.15.).

În cazul în care un parametru efectiv este un nume de tablou, apelul prin valoare devine prin referință, parametrului formal corespunzător lui *i* se atribuie ca valoare adresa primului element al tabloului. Deci, dacă se face apelul:

```
int tab[...];
```

```
...
```

```
f(tab);
```

```
...
```

și *f* are antetul:

```
void f(int v[])
```

atunci la apel *v* are aceeași valoare cu *tab*. Aceasta înseamnă că *tab[0]* și *v[0]* exprimă același lucru, adică valoarea lui *tab[0]*. În general, *tab[i]* și *v[i]* au aceeași valoare și anume valoarea elementului de indice *i* al tabloului *tab*.

Folosind pointeri și în alte cazuri, putem să transformăm apelul prin valoare în apel prin referință. Astfel, dacă *x* este o variabilă simplă, atunci noi putem să transferăm, la un apel, în locul valorii lui *x*, adresa lui *x*:

```
int x;
```

```
...
```

```
h(x); /* se transfera valoarea lui x */
```

```
...
```

```
g(&x); /* se transfera adresa lui x */
```

```
...
```

În acest caz cele două funcții vor avea antete diferite și anume:

```
void h(int i)
```

și

```
void g(int *pi)
```

În cazul funcției *g* parametrul formal *pi* este pointer spre date de tip *int*

La apel, lui *pi* i se atribuie ca valoare adresa lui *x*. De aceea, funcția *g* are posibilitatea să modifice valoarea lui *x*. De exemplu, dacă în corpul funcției *g* se utilizează atribuirea:

```
...
*pi=100;
...
```

atunci valoarea 100 se atribuie variabilei *x*, a cărei adresă s-a atribuit la apel lui *pi*.

În felul acesta, apelul prin valoare se poate folosi pentru a realiza apelul prin referință.

În principiu, în limbajul C, apelul prin referință se poate realiza dacă parametrul efectiv are ca valoare o adresă, iar parametrul formal corespunzător este un *pointer*.

Ulterior o să vedem că în limbajul C++ a fost introdus apelul prin referință și el există împreună cu cel prin valoare.

### Exerciții:

- 8.1 Să se scrie o funcție care dintr-o dată calendaristică definită prin numărul zilei din an și anul respectiv, determină luna și ziua din luna respectivă.

Această funcție a fost definită în exercițiul 6.7. Ea a avut trei parametri:

<i>zz</i>	- Ziua din an.
<i>an</i>	- Anul.
<i>tzll</i>	- Tablou de tip <i>int</i> de 2 elemente.

Funcția atribuie ziua determinată, elementului *tzll*[0], iar luna elementului *tzll*[1].

În funcția de mai jos se înlocuiește parametrul *tzll* cu doi parametri de tip pointer.

### FUNCȚIA BVIII1

```
void pluna_si_ziua( int zz, int an, int *zi,
                  int *luna)
/* determina luna si ziua din luna;
   zz - ziua din an;
   an - anul;
   zi - pointer a carui valoare este adresa zonei in care
```



se pastreaza ziua determinata de functie  
 luna - pointer a carui valoare este adresa zonei in care  
 se pastreaza luna determinata de functie \*/

```
{
    int bisect = an %4 == 0 && an%100 || an%400 == 0;
    int i;
    extern int nrzile[];

    for(i = 1; zz > nrzile[i] + (i==2 && bisect); i++)
        zz -= (nrzile[i] + (i==2 && bisect));

    *zi = zz; /* pastreaza valoarea lui zz in zona de memorie alocata
               parametrului efectiv corespunzator parametrului
               formal zi */

    *luna = i; /* pastreaza valoarea lui i in zona de memorie alocata
                parametrului efectiv corespunzator parametrului
                formal luna */
}
```

- 8.2 Să se scrie o funcție care afișează caracterele unui tablou și citește un întreg de tip *int*.

Funcția are doi parametri:

- |             |   |
|-------------|---|
| <i>text</i> | - Tablou unidimensional de tip caracter și care are aceeași utilizare ca parametrul <i>text</i> din funcția 6.3.                  |
| <i>x</i>    | - Pointer spre întregi de tip <i>int</i> ; are ca valoare adresa zonei de memorie în care se păstrează valoarea întregului citit. |

Această funcție este asemănătoare cu funcția definită în exercițiul 6.3. Diferența constă în aceea că, funcția definită în exercițiul 6.3. atribuie numărul citit variabilei globale *v\_int*, în loc să-l transfere funcției care face apelul, ca în cazul de față.

Funcția de față returnează aceleași valori ca și cea amintită mai sus, adică 0 la întâlnirea sfârșitului de fișier și 1 altfel.

## FUNCȚIA BVIII2

```
int pcit_int(char text[], int *x)
/* - citește un întreg și-l pastrează în zona de memorie a carei adresa este
   valoarea lui x;
   - returnează:
```

```

        0 - la intilnirea sfirsitului de fisier;
        1 - altfel */
{
    char t[255];

    for ( ; ; ) {
        printf(text);
        if(gets(t) == NULL)
            return 0;
        if(sscanf(t,"%d", x) == 1)
            return 1;
    }
}

```

#### Observație:

Deoarece *x* are ca valoare chiar adresa zonei de memorie în care se păstrează întregul convertit din ASCII în *int* prin funcția *sscanf*, în apelul acestei funcții se folosește ca parametru efectiv chiar *x*. El nu mai trebuie să fie precedat de operatorul adresă (&).

8.3 Să se scrie o funcție care citește un întreg de tip *int* care aparține unui interval dat.

Ea are parametri:

<i>text</i>	- Tablou unidimensional de tip <i>char</i> care are aceeași utilizare ca în funcția definită în exercițiul 6.4. (funcția <i>cit_int_lim</i> ).
<i>inf</i>	- Întreg de tip <i>int</i> care are aceeași utilizare ca în funcția <i>cit_int_lim</i> .
<i>sup</i>	- Întreg de tip <i>int</i> care are aceeași utilizare ca în funcția <i>cit_int_lim</i> .
<i>pint</i>	- Pointer spre întregi de tip <i>int</i> ; are ca valoare adresa zonei de memorie în care se păstrează numărul citit.

Ca și funcția *cit\_int\_lim*, funcția de față returnează:

0	- La intilnirea sfirsitului de fisier.
1	- Altfel.

#### FUNCȚIA BVIII3

```
int pcit_int(char [], int *); /* prototip */
```

```

int pcit_int_lim(char text[], int inf,
                 int sup, int *pint)
/* - citește un întreg de tip int ce aparține intervalului [inf,sup] și-l
   păstrează în zona de memorie a carei adresă este valoarea paramet-
   rului pint;
   - returnează:
       0 - la întâlnirea sfârșitului de fișier;
       1 - altfel. */
{
    for( ; ; ) {
        if(pcit_int(text, pint) == 0)
            return 0; /* s-a întâlnit EOF */
        if(*pint >= inf && *pint <= sup)
            return 1;
        printf("întregul tastat nu aparține\
               intervalului:");
        printf("[%d,%d]\n", inf,sup);
        printf("se reia citirea\n");
    }
}

```

8.4 Să se scrie o funcție care citește o dată calendaristică compusă din zi, luna și an.

Funcția validează data calendaristică respectivă. Ea are trei parametri de tip pointer:

- |              |  |
|--------------|--|
| <i>pzi</i>   | - Are ca valoare adresa zonei de memorie în care se păstrează numărul zilei. |
| <i>pluna</i> | - Are ca valoare adresa zonei de memorie în care se păstrează numărul lunii. |
| <i>pan</i>   | - Are ca valoare adresa zonei de memorie în care se păstrează anul.          |

Funcția returnează:

- |   |  |
|---|--|
| 0 | - Dacă s-a întâlnit sfârșitul de fișier. |
| 1 | - Altfel.                                |

Pentru validarea datei calendaristice se apelează funcția *v\_calend* definită în exercițiul 6.5.

Funcția de față este analogă cu cea definită în exercițiul 6.8 Diferența constă în utilizarea parametrilor și a funcției care citește cele 3 valori din compunerea datei calendaristice. Astfel, funcția de față utilizează trei

parametri de tip pointer spre *int*, pentru transferul celor trei valori citite, spre deosebire de funcția *cit\_data\_calend* definită în exercițiul 6.8, care utilizează în același scop un tablou de tip *int*. De asemenea, funcția de față apelează funcția *pcit\_int\_lim* pentru a citi cele trei valori ale datei calendaristice, spre deosebire de funcția *cit\_data\_calend* care utilizează funcția *cit\_int\_lim*.

## FUNCȚIA BVIII4

```
int pcit_int_lim( char [], int,
                 int, int *); /* prototip */
```

```
int pcit_data_calend( int *pzi, int *pluna, int *pan)
/* - citește o dată calendaristică, o validează și o păstrează în zonele de
   memorie a căror adrese sînt definite de cei trei parametri formali
   de tip pointer;
```

- funcția returnează:

0 - la întâlnirea sfîrșitului de fișier;

1 - altfel. \*/

```
{
  static char ziua[] = "ziua: ";
  static char luna[] = "luna: ";
  static char an[] = "anul: ";
  static char er[] = "s-a tastat EOF";
```

```
  for( ; ; ) {
```

```
    /* se citește ziua */
```

```
    if(pcit_int_lim(ziua, 1, 31, pzi) == 0) {
      printf("%s\n", er);
      return 0;
    }
```

```
    /* se citește luna */
```

```
    if(pcit_int_lim(luna, 1, 12, pluna) == 0) {
      printf("%s\n", er);
      return 0;
    }
```

```
    /* se citește anul */
```

```
    if(pcit_int_lim(an, 1600, 4900, pan) == 0) {
      printf("%s\n", er);
      return 0;
    }
```

```

    }

/* validare data calendaristica */
    if(v_calend(*pzi, *pluna, *pan))
        return 1;
    printf("data calendaristica este eronata\n");
    printf("se reia citirea datei\
        calendaristice\n");
}
}

```

- 8.5 Să se scrie un program care citește o dată calendaristică și afișează data calendaristică pentru ziua următoare.

Acest program este analog cu cel definit în exercițiul 6.9. Programul de față utilizează funcții ce au ca parametri pointeri, spre deosebire de cel definit în exercițiul 6.9., care apelează funcții ce utilizează variabila globală `v_int` și parametrii de tip tablou.

## PROGRAMUL BVIII5

```

#include <stdio.h>
#include <stdlib.h>
#include "bviii2.cpp" /* functia pcit_int */
#include "bviii3.cpp" /* functia pcit_int_lim */
#include "bvi5.cpp" /* functia v_calend */
#include "bvi6.cpp" /* functia zi_din_an */
#include "bviii1.cpp" /* functia pluna_si_ziua */
#include "bviii4.cpp" /* functia pcit_data_calend */

int nrzile[] = {0, 31, 28, 31, 30, 31, 30, 31,
               31, 30, 31, 30, 31};

main() /* citește o data calendaristica, o validează și în caz că este
        corectă, afișează data calendaristica a zilei următoare */
{
    int zz, ll, aa;

/* citește și validează data calendaristica */
    if(pcit_data_calend(&zz, &ll, &aa) == 0)
        exit(1);
    if(zz == 31 && ll == 12) {

```

```

/* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
    zz = 1;
    ll = 1; /* ianuarie */
    aa++; /* anul urmator */
}
else /* se determina ziua urmatoare */
    pluna_si_ziua(zi_din_an(zz,ll,aa)+1,
                  aa,&zz,&ll);

/* afiseaza data calendaristica a zilei urmatoare */
printf("ziua:%d\tluna:%d\tanul:%d\n",zz,ll,aa);
}

```

### Observație:

Parametrii efectivi *&zz*, *&ll*, *&aa* de la apelul funcției *pcit\_data\_calend* atribuie parametrilor formali corespunzători, adresele zonelor de memorie alocate variabilelor *zz*, *ll*, și respectiv *aa*.

La revenirea din funcția *pcit\_data\_calend* zonele de memorie alocate acestor variabile vor conține valorile corespunzătoare zilei, lunii și anului care au fost citite prin apelul respectiv (dacă nu s-a tastat sfârșitul de fișier).

### 8.6 Să se scrie o funcție care citește:

- valoarea variabilei *m* de tip *int*;
- valoarea variabilei *n* de tip *int*;
- $m \times n$  numere care reprezintă elementele unei matrice de ordinul  $m \times n$ .

Funcția are următorii parametri:

<i>dmat</i>	- Tablou unidimensional de tip <i>double</i> în care se păstrează elementele matricei prin liniarizare.
<i>max</i>	- Maximul produsului $m \times n$ admis.
<i>nrlin</i>	- Pointer a cărui valoare este adresa zonei de memorie în care se păstrează valoarea lui <i>m</i> .
<i>nrcol</i>	- Pointer a cărui valoare este adresa zonei de memorie în care se păstrează valoarea lui <i>n</i> .

Această funcție este asemănătoare cu funcția *gdcimat* definită în exercițiul 5.1. Diferența dintre ele constă în faptul că funcția *gdcimat* nu utilizează parametri de tip pointer pentru a transfera valorile lui *m* și *n* în

afara funcției, ci două variabile globale.

## FUNCȚIA BVIII8

```
int ndcit(int, double []); /* prototip */

int pdcitmat(double dma[],int max,
              int *nrlin,int *nrcol)
/* - citește pe:
   m-numar de linii;
   n-numar de coloane;
   - m*n numere de tip double pe care le pastreaza in matricea dma
   prin liniarizare;
   - returneaza valoarea m*n;
   m - se pastreaza in zona de memorie definita de
       parametrul nrlin;
   n - se pastreaza in zona de memorie definita de
       parametrul nrcol. */
{
    int i;
    char t[255];
    char er[]="s-a tastat EOF\n";

    do { /* se citesc valorile lui m si n */
        do { /* citește pe m */
            printf("numarul de linii= ");
            if(gets(t) == NULL ) {
                printf(er);
                exit(1);
            }
            if(sscanf(t,"%d", nrlin)==1 &&
                *nrlin > 0 && *nrlin <= max)
                break;
            printf("nu s-a tastat un intreg in\
                intervalul [1,%d]\n", max);
        } while(1);

        do { /* citește pe n */
            printf("numarul de coloane= ");
            if(gets(t) == NULL ) {
                printf(er);
                exit(1);
            }
        }
    }
```

```

        if(secanf(t,"%d",nrcol)==1 &&
           *nrcol > 0 && *nrcol <= max)
            break;
        printf("nu s-a tastat un intreg in\
              intervalul [1,%d]\n", max);
    } while (1);

    i = *nrlin * *nrcol;
    if(i <=max)
        break;
    printf("produsul m*n=%d depaseste pe\
           max=%d\n", i, max );
    printf("se reiau citirile lui n si n\n");
} while(1);

/* se citesc elementele matricei tastate pe linii */
if(ndcit(i,dmat) != i ) {
    printf("nu s-au tastat %d elemente\n", i);
    exit(1);
}
return i;
}

```

#### Observații:

1. Funcția *ndcit* este definită în exercițiul 4.37.
  2. Citirea numerelor *m* și *n* se poate realiza mai simplu folosind funcția *pcit\_int\_lm* definită în exercițiul 8.3.
- 8.7 Să se scrie un program care citește elementele de tip *double* ale două matrice *a* și *b*, calculează produsul lor și-l afișează.

Dacă

$c = a \cdot b$

atunci

$c[i,j] = a[i,0] \cdot b[0,j] + a[i,1] \cdot b[1,j] + \dots + a[i,n-1] \cdot b[n-1,j]$

pentru

$i = 0, 1, \dots, m-1; j = 0, 1, \dots, s-1$

unde:



- a* - Este o matrice de ordinul  $m \times n$ .  
*b* - Este o matrice de ordinul  $n \times s$ .

Matricea produs  $c$  este de ordinul  $m \times s$ .

## PROGRAMUL BVIII7

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "bviii6.cpp" /* functia pdcitmat */

#define MAX 900

main() /* citeste elementele a doua matrice, calculeaza si afiseaza
       matricea produs, cite 4 elemente pe o linie */
{
    int i,j,k;
    int m,n,p,s,r,q;
    int mn,np;
    double a[MAX],b[MAX],c[MAX];

    /* citeste matricea a */
    mn = pdcitmat(a,MAX, &m, &n);

    /* citeste matricea b */
    np = pdcitmat(b,MAX,&p,&s);
    if(n != p) {
        printf("numarul coloanelor matricei\
               a = %d \n",n);
        printf("difera de numarul liniilor ");
        printf("matricei b = %d \n",p);
        exit(1);
    }

    /* se realizeaza produsul c = a*b */
    for(i=0; i < m; i++) {
        q = i*n;
        for(j=0; j < s; j++) {
            r = i*s + j;
            c[r] = 0.0;
            for(k=0; k < n; k++) {
                c[r] += a[q+k]* b[k*s+j];
            }
        }
    }
}
```

```

    }
}

/* afiseaza elementele matricei produs */
printf("\n\n\t\tmatricea produs\n");
k=1;
for(i=0;i<m;i++) {
    p = i*s;
    for(j=0;j<s;j++) {
        printf("c[%d,%d]=%8g ",i,j,c[p+j]);
        if(j%4 == 3) {
            /* afiseaza 4 elemente pe un rind */
            printf("\n");
            k++; /* numara rindurile */
        }
        if(k==23) {
            printf("actionati o tasta pentru a\
continua\n");
            getch();
            k=1;
        }
    }
    printf("\n");
    k++;
}
}
}

```

### 8.3. Legătura dintre pointeri și tablouri

*Numele unui tablou este un pointer deoarece el are ca valoare adresa primului său element. Totuși există o diferență între numele unui tablou și o variabilă de tip pointer. Unei variabile de tip pointer i se atribuie valori la execuție, în timp ce aceasta nu este posibil să se realizeze pentru numele unui tablou. Acesta tot timpul are ca valoare adresa primului său element. De aceea, se obișnuiește să se spună că numele unui tablou este un *pointer constant*.*

**Exemplu:**

```

int t[10];
int *p;
int x;
...

```

```
p=t; /* în urma acestei atribuiri p are aceeași valoare ca și t,  
      adică adresa lui t[0] */
```

```
x=t[0];
```

și

```
x=*p;
```

au același efect: atribuie lui *x* valoarea elementului *t[0]*.

Un parametru formal ce corespunde unui parametru efectiv care este un nume de tablou unidimensional, poate fi declarat fie ca *tablou*, fie ca *pointer* spre tipul tabloului.

Fie declarația:

```
int tab[10];
```

și apelul:

```
f(tab);
```

Funcția *f* poate avea unul din următoarele antete:

```
void f(int t[])
```

sau

```
void f(int *t)
```

Într-adevăr, declarația:

```
int t[]
```

definește pe *t* ca nume de tablou, iar parametrului *t* i se atribuie la apel valoarea lui *tab*, adică adresa lui *tab[0]*. În felul acesta, construcțiile *tab[i]* și *t[i]* reprezintă unul și același element al *i + 1*-lea al tabloului *tab*.

Parametrul formal *t* declarat în acest fel este deci un pointer (are ca valoare o adresă) spre *int*. Spre deosebire de *tab*, *t* este chiar un pointer variabil. El este alocat pe stivă și lui i se atribuie o valoare prin apelul funcției. Ulterior o să vedem că noi putem să modificăm valoarea lui *t*. Nu același lucru se poate spune despre *tab*. Acesta este un pointer constant, a cărui valoare nu poate fi schimbată la execuție.

Din cele de mai sus rezultă că parametrul *t* este un pointer variabil spre *int* și deci el poate fi declarat printr-o declarație de forma:

```
int *t
```

În concluzie, dacă un parametru formal corespunde unui parametru efectiv care este nume de tablou unidimensional, atunci el poate fi declarat fie ca tablou:

*tip nume\_parametru\_formal[]*

fie ca pointer:

*tip \*nume\_parametru\_formal*

Cele două declarații sînt echivalente și de aceea, ambele declarații permit ca în corpul funcției să se utilizeze construcția:

*nume\_parametru\_formal[indice]*

## 8.4. Operații cu pointeri

Asupra pointerilor se pot face diferite operații pe care le precizăm în continuare.

### 8.4.1. Operații de incrementare și decrementare a pointerilor

Operatorii ++ și -- se pot aplica la operandul de tip *pointer*. Ei se execută altfel decît asupra datelor care nu sînt pointeri.

Operatorul de incrementare (++) aplicat unui operand de tip *pointer* spre tipul *t*, mărește adresa, care este valoarea operandului, cu numărul de octeți necesari pentru a păstra o dată de tip *t*.

Operatorul de decrementare are un efect similar, diferența constînd în aceea că în acest caz valoarea operandului se micșorează cu numărul de octeți necesari pentru a păstra o dată de tipul spre care pointează operandul.

De exemplu, dacă avem declarația:

`int *p;`

atunci expresiile:

`++p`

și

`p++`

măresc valoarea lui *p* cu 2, deoarece o dată de tip *int* se păstrează pe 2 octeți.

În mod analog, expresiile:

`--p`

și

**p--**

micșorează valoarea lui *p* cu 2.

Aceste operații sînt utile cînd se au în vedere prelucrări de date de tip tablou.

**Exemplu:**

```
double tab[10];  
double *p;  
int i;
```

Fie:

```
p = &tab[i];
```

unde  $0 < i < n - 1$ . Instrucțiunile expresie.

```
p++; și ++p;
```

mărește valoarea lui *p* cu 8, deci *p* va avea ca valoare adresa următoare elementului *tab[i]*, adică adresa elementului *tab[i + 1]*

În mod analog, instrucțiunile expresie

```
p--; și --p;
```

micșorează valoarea lui *p* cu 8, deci *p* va avea ca valoare adresa elementului precedent lui *tab[i]*, adică adresa elementului

```
tab[i-1].
```

### 8.4.2. Adunarea și scăderea unui întreg dintr-un pointer

Dacă *p* este un pointer spre tipul *t* și *n* un întreg, atunci se pot utiliza expresiile:

```
p+n
```

și

```
p-n
```

Expresia:

```
p+n
```

are ca valoare, valoarea lui *p* mărită cu produsul *r\*n*, unde prin *r* am notat numărul de octeți necesari pentru a păstra în memorie o dată de tipul *t*.

În mod analog, valoarea expresiei

```
p-n
```

este valoarea lui  $p$  micșorată cu produsul  $r \cdot n$ .

Fie  $tab$  un tablou unidimensional de tip  $t$ . Atunci  $tab$  este un pointer (constant) și deci o expresie de forma:

$tab + n$

este corectă. Conform celor spuse mai sus, rezultă că această expresie este chiar adresa elementului  $tab[n]$ . Într-adevăr,  $tab$  este adresa lui  $tab[0]$ . Atunci  $tab + 1$  are ca valoare adresa lui  $tab[0]$  mărită cu  $r$ ,  $r$  fiind numărul de octeți alocați pentru o dată de tip  $t$ , adică numărul de octeți ocupați de elementul  $tab[0]$ .

Deci  $tab + 1$  are ca valoare adresa elementului  $tab[1]$ .

În general,  $tab + n$  va avea ca valoare adresa elementului  $tab[n]$ .



Deoarece  $tab + n$  este adresa elementului  $tab[n]$ , rezultă că

$*(tab + n)$

are ca valoare chiar valoarea elementului  $tab[n]$ . Deci, o atribuire de forma:

$x = tab[n]$

este echivalentă cu:

$x = *(tab + n)$

În general, *variabilele cu indici* se pot înlocui prin *expresii cu pointeri*. Astfel de înlocuiri conduc la optimizări înlocuindu-se operațiile de înmulțire care intervin la evaluarea variabilelor cu indici, prin adunări. De aceea, se recomandă utilizarea expresiilor cu pointeri în locul variabilelor cu indici.

### 8.4.3. Compararea a doi pointeri

Doi pointeri care pointează spre elementele aceluiași tablou pot fi comparați folosind operatorii de relație și de egalitate.

Astfel, dacă  $p$  pointează spre elementul  $t[i]$  al tabloului  $t$  (adică are ca valoare adresa elementului  $t[i]$ ) și  $q$  spre elementul  $t[j]$  al aceluiași tablou, atunci expresiile:

$p < q$ ,  $p \leq q$ ,  $p \geq q$ ,  $p > q$ ,  $p == q$  și  $p != q$

sînt legale. De exemplu, expresia:

$p < q$

are valoarea *adevărat* (1), dacă  $i < j$  și *fals* (0) în caz contrar.

În mod analog se evaluează și celelalte expresii de mai sus.

Operatorii de egalitate ( $==$  și  $!=$ ) pot fi folosiți pentru a compara pointerii cu o constantă specială *NULL*. Aceasta este definită în fișierul *stdio.h* astfel:

```
#define NULL 0
```

Ea reprezintă așa numitul *pointer nul*.

Dacă  $p$  este un pointer spre orice tip, atunci se pot face comparații de forma:

$p == \text{NULL}$

și

$p != \text{NULL}$

În limbajul C++ se recomandă să nu se utilizeze constanta *NULL*, ci chiar valoarea ei zero:

$p == 0$

și

$p != 0$

sau echivalentele lor:

$!p$

și

$p$

Într-adevăr,  $p == 0$  are valoarea adevărat atunci și numai atunci cînd  $!p$  are valoarea adevărat.

Același lucru se poate spune și despre celelalte două expresii:

$p != 0$

are valoarea fals atunci și numai atunci când  $p$  are valoarea zero, adică când  $p$  este fals.

În general, dacă un pointer are valoarea zero (pointerul nul), înseamnă că el nu definește o adresă. Așa cum o să vedem mai târziu, astfel de teste sînt uneori necesare pentru a pune în evidență anumite erori.

Ținînd seama de faptul că *NULL* nu se mai utilizează în programarea curentă în limbajul C++, nu o vom utiliza nici la scrierea programelor în C. Pînă în prezent am utilizat constanta *NULL* la apelul funcției *get*. Aceasta returnează pointerul spre zona în care se păstrează șirul de caractere citit. Deoarece la înțilnirea sfîrșitului de fișier, conținutul acestei zone nu este definit, funcția *gers* returnează zero în acest caz.

#### 8.4.4. Diferența a doi pointeri

Doi pointeri care pointează spre elementele aceluiasi tablou pot fi scăzuți. Fie  $p$  un pointer spre elementul  $t[i]$  al tabloului  $t$  și  $q$  un pointer spre elementul  $t[i+n]$  al aceluiasi tablou. Atunci diferența

$$q - t$$

are valoarea  $n$ .

**Exerciții:**

- 8.8 Să se scrie o funcție care citește cel mult  $n$  elemente de tip *double* și le pastrează în zona de memorie a cărei adresă de început este valoarea parametrului formal  $p$  al funcției. Funcția returnează numărul numerelor citite.

Funcția de față este analogă cu funcția *ndcit* definită în exercițiul 4.37. În acest caz se utilizează expresii cu pointeri în locul indicilor.

#### FUNCȚIA BVIII8

```
int pndcit(int n, double *p)
/* - citește cel mult n numere si le pastreaza in zona spre care pointeaza
   p;
   - returneaza numarul numerelor citite. */
{
    double d;
    char t[255];
    int i = 0;
    double *q = p+n;

    while( p < q ) {
```



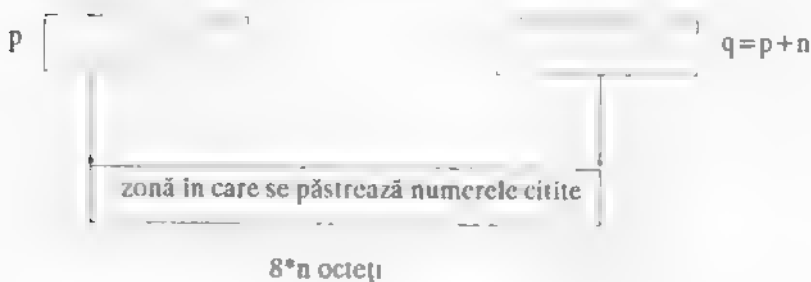
```

printf("elementul [%d]= ", i);
if (gets(t) == NULL )
    return i;
if (scanf(t, "%lf", &d) != 1)
    break;
*p++ = d;
i++;
}
return i;
}

```

### Observații:

1. Parametrul  $p$  are ca valoare la apel, adresa de început a zonei în care se păstrează numerele citite. Dimensiunea acestei zone este de  $8 \cdot n$  octeți, pentru a putea păstra cel mult  $n$  numere de tip *double*.



2. Instrucțiunea:

```
*p++ = d;
```

memorează numărul citit în zona spre care pointează  $p$

Totodată pointerul  $p$  se incrementează, deci valoarea lui se mărește cu 8. În felul acesta, la iterația următoare,  $p$  pointează spre zona în care urmează a fi memorat elementul următor.

După păstrarea a  $n$  numere,  $p$  devine egal cu valoarea lui  $q$ . Deci ciclul *while* în acest moment trebuie să se termine, dacă nu cumva s-a terminat înainte, deoarece s-au tastat mai puțin de  $n$  numere.

- 8.9 Să se scrie o funcție care citește componentele unui vector precedate

de numărul lor. Funcția returnează numărul componentelor vectorului respectiv.

Numărul componentelor se citește apelînd funcția *pcit\_int\_lim* definită în exercițiul 8.3.

Componentele vectorului se citesc apelînd funcția *pdvcit* definită în exercițiul 8.8.

## **FUNCȚIA BVIII9**

```
int pdvcit(int nmax, double dvector[])
/* - citește un întreg n și cele n componente ale vectorului dvector;
   - returnează valoarea lui n. */
{
    int i,n;
    char t[255];

    if(pcit_int_lim("numarul componentelor=",1,
                   nmax,&n) == 0 ) {
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if((i=pdvcit(n,dvector)) != n ) {
        printf("nu s-a reusit citirea celor n=%d\
               componente\n", n );
        printf("s-au citit numai %d componente\n",i);
    }
    return i;
}
```

### **Observație:**

Funcția *pcit\_int\_lim* a fost apelată folosind șirul de caractere:

"numarul componentelor="

ca primul parametru efectiv al apelului. Acest parametru este păstrat de compilator într-o zonă specială rezervată șirului de caractere, iar la apel, se atribuie parametrului formal corespunzător, adresa de început a zonei în care se păstrează șirul respectiv.

8.10 Să se scrie un program care citește componentele vectorilor  $x$  și  $y$ , calculează și afișează produsul lor scalar.

Componentele celor doi vectori sînt precedate de numărul lor.

Programul este similar cu cel definit în exercițiul 4.39. În cazul de față se folosesc expresii cu pointeri în locul variabilelor cu indici.

## PROGRAMUL BVIII10

```
#include <stdio.h>
#include <stdlib.h>

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bviii8.cpp" /* pndcit */
#include "bviii9.cpp" /* pdvcit */

#define MAX 1000

main() /* - citește componentele vectorilor x și y precedate de
        numărul lor;
        - calculează și afișează produsul lor scalar. */
{
    int i,n;
    double *p,*q;
    double x[MAX],y[MAX],s;

    /* citește pe n și componentele vectorului x */
    n = pdvcit(MAX,x);

    /* citește componentele vectorului y */
    if(pndcit(n,y) != n) {
        printf("nu sînt n=%d componente pentru\
              y\n",n);
        exit(1);
    }

    /* calculează produsul scalar */
    s = 0.0;
    for(p=x,q=y, i=0; i < n; i++,p++,q++)
        s += *p * *q;
    printf("(x,y)= %g\n", s);
}
```

## 8.5. Modificatorul const

Am văzut în capitolul 1 că o constantă se poate defini prin caracterele care o compun.

Într-adevăr, caracterele din compunerea unei constante definesc atât tipul cit și valoarea acesteia.

Exemple:

1234	constantă de tip <i>int</i>
1.5	constantă de tip <i>double</i>
'a'	constantă caracter
"a"	constantă șir

Tot în capitolul 1 s-au definit *constantele simbolice*. O astfel de constantă se definește cu ajutorul construcției *#define* tratată de preprocesor. Ea are următorul format:

**#define** *nume succesiune\_de\_caractere*

Printr-o astfel de construcție se atribuie un nume unei constante.

La preprocesare, *nume* se înlocuiește peste tot prin *succesiune\_de\_caractere*, exceptând cazurile cînd *nume* se află într-un comentariu sau șir de caractere.

La compilare, *nume* definit printr-o construcție *#define* nici nu există în afara șirurilor de caractere și a comentariilor. De aceea, valoarea lui nici nu poate fi modificată la execuția programului.

În limbajele C și C++ există posibilitatea de a defini date constante folosind modificatorul *const* în declarații. Printr-o astfel de declarație, unui nume *i* se poate atribui o valoare inițială care nu poate fi modificată, în mod obișnuit, printr-o expresie de atribuire, ca în cazul variabilelor. De aceea, o dată declarată cu ajutorul modificatorului *const* se consideră că este o *constantă*.

Formatele posibile ale unei declarații cu modificatorul *const* sînt:

```
tip const nume = valoare;  
tip const *nume = valoare;  
const tip nume = valoare;  
const tip *nume;  
const nume = valoare;
```

O declarație de forma celor de mai sus se spune că este o *declarație de constantă*.

O declarație de constantă de forma:

(1) *tip const nume = valoare;*

este asemănătoare cu o declarație de variabilă obișnuită de forma:

(2) *tip nume = valoare;*

Diferența dintre cele două declarații constă în aceea că valoarea lui *nume* atribuită prin declarația (1) nu poate fi schimbată folosind o expresie de atribuire de forma:

*nume = expresie*

în timp ce aceasta este posibil pentru *nume* declarat prin declarația (2).

**Exemple:**

1. `int const i = 10;`

*i* este o constantă care are valoarea 10. O expresie de atribuire de forma  
*i = 3*

este eronată.

Constanta *i* declarată ca mai sus diferă față de o constantă definită prin `#define`, deoarece ea nu este prelucrată de preprocesor. Numele ei există la compilare.

O expresie de forma:

*y = i+7*

este corectă și va atribui lui *y* valoarea 17.

2. `double const pi = 3.14159265;`

declară numele *pi* ca fiind o constantă de tip *double* și care are valoarea 3.14159265.

O atribuire de forma:

*pi = 3.1415*

generează o eroare la compilare.

*pi* poate fi utilizat în expresii de forma:

*pi\*r\*r*  
*sin(pi/2+x)*  
*cos(x-pi)*

3. `char *const s = "sir";`

Aici *tip* este *char \**, deci *s* este un *pointer constant* spre zona în care se păstrează șirul de caractere format din literele *s*, *i*, *r* și caracterul *NUL*. În acest caz valoarea lui *s* nu poate fi schimbată. El fiind *pointer*, are ca valoare o adresă a unei zone de memorie de dimensiune egală cu 4 octeți. În această zonă se păstrează șirul de caractere indicat mai sus. Conținutul acestei zone poate fi modificat. Astfel, în timp ce o atribuire de forma *s = t*; unde *t* este un *pointer* spre caractere, nu este acceptată de compilator, atribuirile:

```
*s = '1';
*(s+1) = '2';
```

sînt corecte. Cu ajutorul lor se schimbă caracterul *s* cu 1 și *i* cu 2 în zona spre care pointează *s*.

Declarația:

(3) *tip const \*nume = valoare*;

definește pe *nume* ca un *pointer* spre o zonă constantă. În acest caz, valoarea pointerului *nume* se poate schimba. De exemplu, dacă se consideră declarația:

```
char const *s = "sir";
```

atunci atribuirea:

```
s = t;
```

unde *t* este un *pointer* spre *char* este corectă. În schimb atribuirile:

```
*s = 'a'
*(s+1) = 'b'
```

sînt eronate, deoarece *s* pointează spre o zonă în care se păstrează o dată constantă. Cu toate acestea, constanta respectivă poate fi modificată folosind un *pointer* diferit de *s*:

```
char *p;
```

```
p = (char *)s; /* p, ca și s, pointeaza spre zona în care se pastreaza
                sirul de caractere "sir" */
```

```
*p = 'a';
*(p+1) = 'b';
```

Aceste atribuiri sînt corecte, deoarece *p* nu mai este un *pointer* spre o zonă constantă.

Declarația:

(4) *const tip nume = valoare*;

este identică cu declarația (1) dacă *tip* nu este un tip pointer.

Dacă *tip* este un tip pointer, adică declarația (4) este de forma:

(5) *const tip \*nume = valoare;*

atunci ea este identică cu declarația (3) de mai sus. De obicei, se utilizează formatul (5), în locul formatului (3) pentru a declara un pointer spre o zonă constantă (a cărei valoare nu poate fi modificată direct, adică prin atribuire în care se folosește *nume*:

*\*nume = ...*

*\*(nume+k)=...).*

Declarația

(6) *const tip \*nume*

se utilizează pentru a declara un parametru formal.

Fie funcția *f* de antet:

*tip f(tip \*nume)*

La apelul funcției *f*, parametrului formal *nume* i se atribuie ca valoare o adresă. Am văzut că acest fapt dă posibilitatea funcției *f* să modifice *data* păstrată în zona de memorie spre care pointează *nume* folosind o atribuire de forma:

*\*nume = valoare*

Există cazuri în care funcția apelată în acest fel nu are voie să modifice *data* din zona de memorie a cărei adresă este atribuită unui parametru formal. Ea trebuie numai să aibă acces la *data* respectivă. Pentru a proteja *data* față de eventualele atribuiri neautorizate, vom declara parametrul formal respectiv ca un pointer spre o dată constantă. În acest caz, antetul funcției *f* devine:

*tip f(const tip \*nume)*

În acest caz o atribuire de forma:

*\*nume = valoare*

este interzisă. De aceea, declarația:

*const tip \*nume*

se recomandă a fi utilizată pentru orice parametru formal care la apel are ca valoare adresa unei zone de memorie al cărui conținut nu poate fi modificat de funcția apelată.

## 8.6. Funcții standard utilizate la prelucrarea șirurilor de caractere

Biblioteca standard conține o serie de funcții care permit operații cu șiruri de caractere. Majoritatea acestor funcții au prototipul în fișierul *string.h*. Mai jos prezentăm funcțiile mai importante din această clasă.

Un șir de caractere se păstrează într-o zonă de memorie organizată ca tablou unidimensional de tip *char*. Fiecare caracter se păstrează pe cîte un octet prin codul său numeric.

Cel mai frecvent cod utilizat în acest scop este codul ASCII.

După ultimul caracter al șirului se păstrează caracterul *NUL* ("\\0").

Pentru a opera cu un șir de caractere se poate utiliza *numele tabloului* al cărui elemente au ca valori codurile caracterelor șirului respectiv.

Spunem despre acest *nume* ca este un pointer constant spre șirul respectiv.

Evident, se pot utiliza și pointeri variabili spre un șir de caractere.

**Exemplu:**

```
char tab[] = "Acesta este un sir";
```

Șirul de caractere "Acesta este un sir" se păstrează în zona de memorie alocată lui *tab*.

*tab* are ca valoare adresa de început a zonei de memorie în care se păstrează caracterele șirului.

*tab* - Adresa caracterului *A*.

*tab+1* - Adresa caracterului *e*.

*tab+2* - Adresa caracterului *s*.

etc.

*tab[0]* - Codul ASCII al caracterului *A*.

*tab[1]* - Codul ASCII al caracterului *e*.

etc.

*\*tab* - Codul ASCII al caracterului *A*.

*\*(tab+1)* - Codul ASCII al caracterului *e*.

etc.

Un efect similar se obține cu ajutorul declarației:



`char *const p = "Acesta este un sir";`

Șirul de caractere se păstrează într-o zonă de memorie rezervată pentru a păstra șiruri de caractere.

Adresa de început a zonei în care se păstrează șirul de față se atribuie pointerului *p*. Acesta, ca și *tab*, este un pointer constant.

*p* - Adresa caracterului *A*.

*p+1* - Adresa caracterului *c*.

*p+2* - Adresa caracterului *e*.

etc.

*p[0]* sau *\*p* - Codul ASCII al caracterului *A*.

*p[1]* sau *\*(p+1)* - Codul ASCII al caracterului *c*.

etc.

În legătură cu șirurile de caractere se au în vedere operații de următorul fel:

- calculul lungimii unui șir de caractere;
- copierea șirurilor de caractere;
- concatenarea șirurilor de caractere;
- compararea șirurilor de caractere.

Funcțiile standard prin care se realizează aceste operații au fiecare un nume care începe cu prefixul *str* (prescurtare de la *string*).

### 8.6.1. Lungimea unui șir de caractere

Lungimea unui șir de caractere se definește prin numărul de caractere proprii care intră în compunerea șirului respectiv. Caracterul *NUL* este un caracter impropriu și el nu este considerat la determinarea lungimii unui șir de caractere. Prezența lui este însă necesară, deoarece la determinarea lungimii unui șir se numără caracterele acestuia până la întâlnirea caracterului *NUL*.

Funcția pentru determinarea lungimii unui șir de caractere are prototipul:

`unsigned strlen(const char *s);`

**Exemple:**

1.

`char *const p = "Acesta este un sir";`

```

unsigned n;
...
n = strlen(p);

```

Lui *n* i se atribuie valoarea 18 (numărul caracterelor propriu din compunerea șirului spre care pointeaza *p*).

2.

```

char tab[] = "Acesta este un sir";
int n;
n = strlen(tab);

```

Variabila *n* primește aceeași valoare ca în exemplul precedent.

3.

```

int n;
n = strlen("Acesta este un sir");

```

Lui *n* i se atribuie aceeași valoare ca în exemplele precedente.

#### Observație:

Parametrul formal al funcției *strlen* este un pointer spre o dată constantă deoarece funcția *strlen* nu are voie să modifice caracterele șirului pentru care determină lungimea.

### 8.6.2. Copierea unui șir de caractere

Adesea este nevoie să se copieze un șir de caractere din zona de memorie în care se află, într-o altă zonă.

În acest scop se poate folosi funcția de prototip:

```
char *strcpy(char *dest, const char *sursa);
```

Funcția copiază șirul de caractere spre care pointează *sursa* în zona de memorie a cărei adresă de început este valoarea lui *dest*.

Funcția copiază atât caracterele proprii șirului, cât și caracterul *NUL* de la sfârșitul șirului respectiv.

Se presupune că zona de memorie în care se face copierea este destul de mare pentru a putea păstra caracterele copiate. În caz contrar se alterează datele păstrate imediat după zona rezervată la adresa definită de parametrul *dest*.

La revenire, funcția returnează adresa de început a zonei în care s-a

transferat șirul, adică chiar valoarea lui *dest*. Această valoare este pointer spre caractere deci tipul returnat de funcție este:

`char *`

De aceea, prototipul funcției *strcpy* începe prin construcția `char *` (tipul valorii returnate de funcție este pointer spre *char*).

Se observă că parametrul *sursa*, care definește zona în care se află șirul ce se copiază, este declarat prin modificatorul *const*. Aceasta deoarece funcția *strcpy* nu are voie să modifice șirul care se copiază. În schimb, parametrul *dest* nu este declarat cu modificatorul *const* deoarece funcția *strcpy* modifică zona spre care pointează *dest* (în ea se copiază caracterele șirului).

**Exemple:**

1.

```
char tab[] = "Acest sir se copiaza";
char t[sizeof tab]; /* are același număr de elemente ca și tab */
...
strcpy(t, tab); /* șirul păstrat în tab se copiaza în zona alocată
                lui t */
```

2.

```
char t[100];
strcpy(t, "Acest sir se copiaza");
```

3.

```
char *p = "Acest sir se copiaza";
char t[100];
char *q;
q = strcpy(t, p);
```

Șirul păstrat în zona spre care pointează *p* se transferă în zona spre care pointează *t*. Valoarea lui *t* se atribuie lui *q*.

Pentru a copia cel mult *n* caractere ale unui șir dintr-o zonă de memorie în alta, se va folosi funcția de prototip

`char *strncpy(char *dest, const char *sursa, unsigned n);`

Dacă *n* > *lungimea șirului* spre care pointează *sursa*, atunci toate caracterele șirului respectiv se transferă în zona spre care pointează *dest*. Altfel se copiază numai primele *n* caractere ale șirului. În rest, funcția *strncpy* are același efect ca și *strcpy*.

#### Exemplu:

```
char *p = "Acest sir se copiaza truncheat";  
char t[10];  
strncpy(t,p,sizeof t);
```

### 8.6.3. Concatenarea șirurilor de caractere

Bibliotecile limbajelor C și C++ conțin o funcție care permite concatenarea unui șir de caractere la sfârșitul unui alt șir de caractere. Una dintre ele are prototipul:

```
char *strcat(char *dest,const char *sursa),
```

Această funcție copiază șirul de caractere din zona spre care pointează *sursa*, în zona de memorie care urmează imediat după ultimul caracter propriu al șirului spre care pointează *dest*. Se presupune că zona spre care pointează *dest* este suficientă pentru a păstra caracterele proprii celor două șiruri care se concatenează, plus caracterul *NUL* care termină șirul rezultat în urma concatenării.

Funcția returnează valoarea lui *dest*.

#### Exemplu:

```
char tab1[100] = "Limbaajul C++";  
char tab2[] = "este c incrementat";  
strcat(tab1," "); /* concateneaza caracterul spatiu dupa cel  
de al doilea caracter + pastrat in tabloul  
tab1 la initializare */  
strcat(tab1,tab2); /* concateneaza textul pastrat in tab2 la  
initializare, dupa spatiul concatenat prin  
instructiunea precedenta */
```

#### Observație:

Funcția *strcat*, la fel ca funcția *strcpy*, nu trebuie să modifice șirul de caractere spre care pointează *sursa*.

O altă funcție de bibliotecă utilizată la concatenarea de șiruri este funcția *strncat*.

Ea are prototipul:

```
char *strncat(char *dest,const char *sursa,unsigned n);
```

În acest caz se concatenează, la sfârșitul șirului spre care pointează *dest*, cel mult *n* caractere ale șirului spre care pointează *sursa*.

Dacă  $n >$  lungimea șirului spre care pointează *sursa*, atunci se concatenează întregul șir, altfel numai primele  $n$  caractere ale acestuia.

**Exemplu:**

```
char tab1[100] = "Limbajul E este mai bun decit ";  
char tab2[] =  
    "limbajul C++ care este un superset a lui C";  
strncat(tab1, tab2, 12);
```

După revenirea din funcție, tabloul *tab1* conține succesiunea de caractere:

Limbajul E este mai bun decit limbajul C++

### 8.6.4. Compararea șirurilor de caractere

Șirurile de caractere se pot compara folosind codurile ASCII ale caracterelor din compunerea lor.

Fie *s1* și *s2* două tablouri unidimensionale de tip caracter folosite pentru a păstra, fiecare, câte un șir de caractere.

Șirurile păstrate în aceste tablouri sînt *egale* dacă au lungimi egale și  $s1[i] = s2[i]$  pentru toate valorile lui *i*.

Șirul păstrat în tabloul *s1* este *mai mic* decît cel păstrat în *s2*, dacă există un indice *i*, așa încît:

$s1[i] < s2[i]$

și

$s1[j] = s2[j]$  pentru  $j=0, 1, \dots, i-1$ .

Șirul păstrat în tabloul *s1* este *mai mare* decît cel păstrat în *s2*, dacă există un indice *i*, așa încît:

$s1[i] > s2[i]$

și

$s1[j] = s2[j]$  pentru  $j=0, 1, \dots, i-1$ .

Compararea șirurilor de caractere se poate realiza folosind funcții standard de felul celor de mai jos.

O funcție utilizată frecvent în compararea șirurilor este funcția de prototip:

```
int strcmp(const char *s1, const char *s2);
```

Notăm cu *sir1* șirul de caractere spre care pointează *s1* și cu *sir2* șirul de caractere spre care pointează *s2* (spunem că un pointer pointează spre un șir dacă valoarea lui este adresa de început a zonei de memorie în care se păstrează șirul respectiv).

Funcția *strcmp* returnează:

- o valoare negativă dacă *sir1* < *sir2*;
- zero dacă *sir1* = *sir2*;
- o valoare pozitivă dacă *sir1* > *sir2*.

O altă funcție pentru compararea a două șiruri este funcția de prototip:

**int strcmp(const char \*s1, const char \*s2, unsigned n);**

Această funcție compară cele două șiruri spre care pointează *s1* și *s2* utilizând cel mult primele *n* caractere din fiecare șir. În cazul în care minimul dintre lungimile celor două șiruri este mai mic decât *n*, funcția *strcmp* realizează aceeași comparație ca și funcția *strcmp*.

Adesea, la compararea șirurilor de caractere dorim să nu se facă distincție între literele mici și mari. Acest lucru este posibil dacă se folosește funcția de prototip:

**int stricmp(const char \*s1, const char \*s2);**

Această funcție returnează aceleași valori ca și funcția *strcmp*, cu deosebirea că la compararea literelor nu se face distincție între literele mari și mici.

**Exemple:**

```
char *sir1 = "ABC";  
char *sir2 = "abc";  
int i;
```

Apelul:

```
i = strcmp(sir1, sir2);
```

returnează o valoare negativă, deoarece literele mari au coduri ASCII mai mici decât literele mici (*A* are codul 65, iar *a* are codul 97), deci

```
"ABC" < "abc"
```

Apelul:

```
i = stricmp(sir1, sir2);
```

returnează zero, deoarece ignorându-se diferența dintre literele mari și mici, cele două șiruri devin egale.

Pentru a limita compararea a două șiruri de caractere la primele cel mult  $n$  caractere ale lor, la comparare ignorându-se diferența dintre literele mici și mari, se va folosi funcția de prototip:

```
int strncmp(const char *s1, const char *s2, unsigned n);
```

**Exerciții:**

8.11 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai lung dintre ele.

Prin *cuvînt* înțelegem o succesiune de caractere diferite de caracterele albe.

Vom presupune că un cuvînt nu are mai mult de 100 de caractere. Cuvintele sînt separate prin caractere albe. La sfîrșit se tastează sfîrșitul de fișer pentru a termina succesiunea de cuvinte.

### PROGRAMUL BVIII11

```
#include <stdio.h>
#include <string.h>

#define MAX 100

main() /* citește o succesiune de cuvinte și-l afișează pe cel mai lung
       dintre ele */
{
    int max=0,i;
    char cuvint[MAX+1];
    char cuvint_max[MAX+1];

    while(scanf("%100s",cuvint) != EOF )
        if(max < (i=strlen(cuvint))) {
            max = i;
            strcpy(cuvint_max,cuvint);
        }
    if(max)
        printf("%s\n", cuvint_max);
}
```

**Observații:**

1. Cuvîntul citit se păstrează în tabloul *cuvint* de MAX+1 elemente. La citire se utilizează specificatorul de format:

%100s

care permite să se citească cel mult 100 de caractere diferite de cele albe. Funcția *scanf* păstrează caracterul NUL după ultimul caracter citit. Deci, un cuvânt de 100 de caractere ocupă 101 octeți.

2. După citirea unui cuvânt se apelează funcția *strlen* pentru a determina numărul caracterelor citite prin *scanf* și păstrate în tabloul *cuvint*.

În acest caz s-a utilizat expresia:

```
i = strlen(cuvint)
```

Apoi se compară lungimea cuvintului citit cu *max*.

Variabila *max* are ca valoare lungimea maximă a cuvintelor citite înaintea celui curent.

Inițial *max* = 0, deoarece nu există nici un cuvânt citit.

Dacă *max* este mai mic decât lungimea cuvintului citit curent, atunci lui *max* i se atribuie această valoare, iar cuvântul citit este transferat în zona de memorie alocată tabloului *cuvant\_max*. În acest scop se apelează funcția *strcpy*:

```
strcpy(cuvant_max,cuvint);
```

- 8.12 Să se scrie un program care citește două cuvinte și le afișează în ordine crescătoare.

Cuvântul se definește ca în exercițiul precedent, adică este o succesiune de cel mult 100 de caractere care nu sînt albe. Cuvintele sînt separate prin caractere albe.

## PROGRAMUL BVIII12

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

main() /* citește două cuvinte și le afișează în ordine crescătoare */
{
    char cuv1[MAX+1];
    char cuv2[MAX+1];

    if (scanf("%100s",cuv1) != 1 ) {
```



```

        printf("nu s-a tastat un cuvint\n");
        exit(1);
    }
    if(scanf("%100s",cuv2) != 1 ) {
        printf("nu s-a tastat un cuvint\n");
        exit(1);
    }
    if(strcmp(cuv1,cuv2) < 0 ) {

/* primul cuvint tastat este mai mic decit cel de-al doilea */
        printf("%s\n",cuv1);
        printf("%s\n",cuv2);
    }
    else {
        printf("%s\n",cuv2);
        printf("%s\n",cuv1);
    }
}

```

8.13 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai mare.

Prin cuvint înțelegem o succesiune de cel mult 100 de caractere care nu sînt albe. Cuvintele sînt separate prin caractere albe. Succesiunea de cuvinte se termină cu sfîrșitul de fișier.

### PROGRAMUL BVIII13

```

#include <stdio.h>
#include <string.h>

#define MAX 100

main() /* citește o succesiune de cuvinte si-l afiseaza pe
        cel mai mare */
{
    char cuvcrt[MAX+1];
    char cuvmax[MAX+1];

    cuvmax[0] = '\0'; /* cuvmax se initializeaza cu cuvintul vid */
    while(scanf("%100s",cuvcrt) != EOF )
        if(strcmp(cuvcrt,cuvmax) > 0 )

/* cuvintul curent este mai mare decit cel pastrat in cuvmax */

```

```

        strcpy(cuvmax,cuvert);
    printf("cel mai mare cuvint este\n");
    printf("%s\n", cuvmax);
}

```

8.14 Să se scrie un program care citește o succesiune de cuvinte, le sortează în ordine crescătoare și apoi le afișează în ordinea respectivă.

Prin *cuvint* înțelegem un șir de litere mici sau mari. La compararea cuvintelor nu se face deosebirea între literele mici și mari. În felul acesta, cuvintele se vor afișa în ordine alfabetică. Vom presupune că lungimea unui cuvint nu depășește 30 de caractere și că sunt cel mult 500 de cuvinte la intrare.

În prima parte se citesc cuvintele și se păstrează în tabloul *tcuvinte*. Acesta este de tip *char*. Fiecare cuvint se termină prin caracterul *NUL*.

La întâlnirea sfârșitului de fișier se trece la sortarea cuvintelor păstrate în tabloul *tcuvinte*.

Sortarea cuvintelor se face folosind metoda bulelor (vezi exercițiul 4.40).

Conform acestei metode, se parcurg elementele tabloului comparându-se de fiecare dată două elemente vecine. Dacă ele nu sunt în ordinea cerută (crescătoare), atunci ele se permută. Permutarea elementelor învecinate este simplă în cazul în care elementele respective sunt numere. În cazul de față elementele sunt cuvinte și permutarea lor este mai complicată. De aceea, vom folosi pointeri spre începutul fiecărui cuvint. Acești pointeri îi păstrăm ca elemente ale unui tablou pe care îl numim *tpointer*.

La terminarea citirii cuvintelor, elementele lui *tpointer* vor pointa spre începuturile cuvintelor păstrate în tabloul *tcuvinte*. Astfel, *tpointer[0]* pointează spre primul cuvint citit și păstrat în *tcuvinte* (are ca valoare adresa de început a zonei de memorie în care se păstrează primul caracter), *tpointer[1]* pointează spre al doilea cuvint și așa mai departe.

**Exemplu:**

Presupunem că se tastează textul:

Programarea orientata spre obiecte este un stil modern  
de programare.

După citirea textului respectiv, tabelele *tpointer* și *tcuvinte* au următoarele conținute:

*tpointer[0]* = adresa lui *tcuvinte[0]*;

Elementele *tcuvinte[0]* - *tcuvinte[11]* conțin caracterele cuvintului

*Programarea*, inclusiv caracterul *NUL* de la sfârșitul cuvintului.

`tpointer[1] = adresa lui tcuvinte[12];`

Elementele `tcuvinte[12]` - `tcuvinte[21]` conțin caracterele corespunzătoare cuvintului *orientara*.

`tpointer[2] = adresa lui tcuvinte[22];`

Elementele `tcuvinte[22]` - `tcuvinte[26]` conțin caracterele corespunzătoare cuvintului *spre* și așa mai departe.

Ultimul element al tabloului *tpointer* la care i s-a atribuit valoare este:

`tpointer[9] = adresa lui tcuvinte[58]`

Elementele `tcuvinte[58]` - `tcuvinte[68]` conțin caracterele corespunzătoare cuvintului *programare*.

Pentru a compara două cuvinte învecinate păstrate în tabloul *tcuvinte*, utilizăm pointeri spre ele, pointeri care sînt memorati în tabloul *tpointer*

Astfel, pentru a compara cuvîntul al *k*-lea din tabloul *tcuvinte*, cu următorul lui, vom apela funcția *strcmp* cu parametri `tpointer[k]` și `tpointer[k+1]`:

```
strcmp(tpointer[k], tpointer[k+1])
```

Dacă la un astfel de apel funcția *strcmp* returnează o valoare pozitivă, înseamnă că `tpointer[k]` pointează spre un cuvînt care este mai mare (urmează în ordine alfabetică) decît cuvîntul spre care pointează `tpointer[k+1]`. În acest caz ar urma să se permute cuvintele respective, deoarece nu sînt în ordinea cerută.

Existența pointerilor spre cuvintele respective face ca să nu mai fie necesară permutarea efectivă a cuvintelor, ci numai a valorilor pointerilor spre ele. Deci, în acest caz se vor permuta valorile pointerilor `tpointer[k]` și `tpointer[k+1]`:

```
if(strcmp(tpointer[k], tpointer[k+1]) > 0)
{
    char *t;

    t = tpointer[k];
    tpointer[k] = tpointer[k+1];
    tpointer[k+1] = t;
}
```

Se observa utilizarea funcției *strcmp*, pentru a realiza comparații între cuvinte la care se ignoră diferența dintre literele mici și mari.

## PROGRAMUL BVIII14

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

#define MAXLC 30
#define MAXNC 500

main() /* sorteaza un sir de cuvinte in ordine crescatoare
      (alfabetica) */
{
    static char tcuvinte[(MAXLC+1)*MAXNC];
    static char *tpointer[MAXNC];
    int c,i,j,k,ind;
    char *t;

    i = 0; /* indice in tpointer */
    j = 0; /* indice in tcuvinte */
    c = getchar();

    /* citirea cuvintelor */
    while(c != EOF) { /* 1 */

        /* avans peste caracterele care nu sînt litere */
        while( !( c >= 'A' && c <= 'Z' ||
                  c >= 'a' && c <= 'z' ) ) {

            /* c nu contine o litera */
            c = getchar();
            if( c == EOF )
                break;
        }
        if( c == EOF )
            break;

        /* c contine o litera; este litera de inceput a cuvintului curent; acest
        cuvint se pastreaza incepind cu elementul tcuvinte[j]; adresa de ince-
        put a cuvintului, adica adresa elementului tcuvinte[j], se pastreaza
        in tabloul tpointer; aceasta adresa este tcuvinte + j si se atribuie
        elementului tpointer[j] */
        tpointer[i++] = tcuvinte + j;
```

```

/* se citesc caracterele cuvintului curent si se pastreaza in tabloul
   tcuvinte incepind cu elementul tcuvinte[j] */
   while( c >= 'A' && c <= 'Z' ||
          c >= 'a' && c <= 'z' ) {
       tcuvinte[j++] = c;
       c = getchar();
   }
/* se pastreaza caracterul NUL dupa cel curent */
   tcuvinte[j++] = '\0';
} /* sfirsit while 1 */

/* s-a terminat citirea cuvintelor */
/* s-au citit i cuvinte */

if( i ) { /* 1 */

/* exista cel putin un cuvint citit */
/* se face sortare */
   ind = 1;
   while ( ind ) {
       ind = 0;
       for( k=0; k < i-1; k++)

/* se analizeaza ordinea cuvintelor vecine */
           if( strcmp(tpointer[k],
                      tpointer[k+1]) > 0 ) {
               /* permutarea pointerilor */
               t = tpointer[k];
               tpointer[k] = tpointer[k+1];
               tpointer[k+1] = t;
               ind = 1;
           } /* sfirsit if */
   } /* sfirsit while */

/* s-a terminat sortarea; se afiseaza cuvintele */
   for( j = 0; j < i; j++ ) {
       printf("%s\n", tpointer[j]);
       if( (j+1)%23 == 0 ) {
           printf("actionati o tasta pentru a\
                  continua\n");
           getch();
       }
   }
}

```

```

    } /* sfirsit for */
} /* sfirsit if 1 */
}

```

## 8.7. Expresie lvalue

În paragraful 3.2.7. s-a definit expresia de atribuire prin următorul format:

(1)  $v = \text{expresie}$

unde:

$v$  - Se consideră că este o variabilă simplă, o variabilă cu indici (permite accesul sau modificarea valorii unui element de tablou) sau definește un element de structură.

Definiția expresiei de atribuire poate fi completată în momentul de față adăugând formatul:

(2)  $*ep = \text{expresie}$

unde:

$ep$  - Este o expresie care definește un pointer nenul și care nu este un pointer spre o zonă constantă. Astfel de expresii s-au utilizat deja în paragrafele precedente.

**Exemplu:**

```

int n;
double x;
void *p;

```

Expresiile de atribuire de forma:

$n = 123, x = 3.14159$

corespund formatului (1), indicat mai sus pentru expresiile de atribuire.

Fie:

$p = \&n;$

atunci am văzut că atribuirea

$*(\text{int } *)p = 123$

realizează același lucru ca și expresia:

`n = 123.`

Expresia:

`(int *)p`

definește un pointer spre o zonă care nu este constantă și deci atribuirea de mai sus corespunde formatului (2) al expresiilor de atribuire. În mod analog, dacă:

`p = &x;`

atunci atribuirea:

`*(double *)p = 3.14159`

realizează același lucru ca și expresia:

`x = 3.14159`

Expresiile utilizabile în *partea stângă* a unei expresii de atribuire se numesc *expresii lvalue*.

Litera *l* provine de la cuvântul englezesc *left*.

Noțiunea de expresie *lvalue* aparține autorilor limbajului C (vezi [2]).

Ulterior s-a introdus și noțiunea de expresie *rvalue*, care este o expresie care se poate utiliza în partea dreaptă a unei expresii de atribuire, dar nu și în partea stângă (vezi [8]).

Din cele de mai sus rezultă că o *expresie lvalue* poate fi:

- un nume de variabilă simplă;
- o variabilă cu indici;  
o construcție ce permite accesul sau modificarea valorii unui element de structură;
- o construcție de forma:

`*ep`

unde:

*ep* - Este o expresie care definește un pointer nenul spre o dată care nu este o constantă.

Fie declarațiile:

`const int *a = 100;`

`int b;`

În acest caz, *a* este un pointer spre o zonă constantă și de aceea, expresia:

**\*a = 123**

nu este corectă. În acest caz **\*a** nu este o expresie *lvalue*. Ea este o expresie *rvalue*, deoarece se poate utiliza în partea dreaptă a expresiilor de atribuire.

Într-adevăr, expresia:

**b = \*a**

este corectă și atribuie variabilei *b* valoarea 100.

## 8.8. Alocarea dinamică a memoriei

Limbajul C permite utilizatorului să aloce date atât pe stivă (date automate), cât și în zone de memorie care nu aparțin stivei (date globale sau statice).

Alocarea datelor pe stivă se face la execuție și ea nu este permanentă. Astfel, dacă declarația:

```
rip nume;
```

se utilizează în corpul unei funcții, atunci variabila *nume* se alocă pe stivă la fiecare apel al funcției respective. La revenirea din funcție, stiva se "curăță" (se reduce la starea avută înaintea apelului) și prin aceasta variabila *nume* nu mai este alocată (devine nedefinită). O alocare de acest fel a memoriei se spune că este *dinamică*.

Pentru datele globale sau statice, memoria este alocată în fazele precedente execuției și alocarea rămâne valabilă până la terminarea execuției programului. De aceea, pentru datele de acest fel se spune că alocarea este *statică* (nu este dinamică).

Limbajele C și C++ oferă utilizatorului posibilitatea de a aloca dinamic memorie și în alt mod decât cel indicat mai sus pentru datele automate. Aceasta se realizează într-o zonă de memorie specială, distinctă de stivă. Această zonă de memorie se numește *memorie heap* (memorie grămadă, morman, de acumulare etc.) Ea poate fi gestionată prin funcții standard în ambele limbaje.

Mai jos indicăm câteva funcții care pot fi utilizate la alocarea dinamică a datelor în memoria *heap*. Aceste funcții sînt comune ambelor limbaje. Mai târziu o să vedem că în limbajul C++ există chiar operatori pentru alocări dinamice în memoria *heap*.

Funcțiile standard pentru gestiunea memoriei *help* au prototipurile în fișierul *alloc.h*.

*Alocarea* unei zone de memorie în memoria *heap* se poate realiza cu



ajutorul mai multor funcții. O funcție utilizată frecvent este funcția *malloc*. Aceasta are prototipul:

```
void *malloc(unsigned n);
```

Funcția alocă în memoria *heap* o zonă contiguă de  $n$  octeți. Ea returnează adresa de început a zonei alocate. Această adresă reprezintă un pointer de tip *void* (*void \**). Prin intermediul acestuia se pot păstra date în zona de memorie alocată în acest fel. Pentru a păstra o dată de un *tip* dat într-o zonă de memorie alocată prin *malloc* este necesar să convertim adresa returnată de funcție spre tipul datei respective.

**Exemplu:**

Se cere să se aloc în memoria *heap* o zonă de memorie pentru a păstra  $n$  valori de tip *int*. În acest scop declarăm un pointer spre tipul *int*:

```
int *p;
```

Apoi, apelăm funcția *malloc* cu ajutorul expresiei de atribuire:

```
p = (int *)malloc(n*sizeof(int))
```

Se observă că valoarea returnată de funcția *malloc* a fost convertită spre tipul *int \**, adică pointer spre *int*.

În continuare putem păstra și utiliza date de tip *int*, folosind pointerul *p*. De exemplu, expresia:

```
*p = 123
```

păstrează întregul 123 în primul doi octeți ai zonei alocate prin expresia de mai sus.

Expresia:

```
y = *p
```

atribuie lui *y* valoarea păstrată mai sus în memoria *heap*.

În general, expresia:

```
*(p+i) = k
```

atribuie valoarea lui  $k$  în zona de memorie de adresă definită prin expresia pointer:

```
p+i
```

Aceeași expresie se poate utiliza pentru a face acces la data respectivă.

**Observații:**

1. Funcția *malloc* are ca parametru un întreg fără semn, adică acesta

trebuie să aparțină intervalului [0,65535].

2. În cazul în care în memoria *heap* nu se poate alocă o zonă de memorie contiguă de atîta octeți cît este valoarea parametrului de la apel, se va returna *pointerul nul*, adică valoarea zero.

De aceea, după apelul funcției *malloc* vom testa valoarea returnată pentru a ne asigura că aceasta nu este zero.

Zonele alocate prin funcția *malloc* pot fi eliberate, pentru a putea fi eventual realocate, folosind funcția standard *free*. Aceasta are prototipul:

```
void free(void *p);
```

Prin apelul ei, se eliberează zona de memorie din memoria *heap*, spre care pointează *p*. Menționăm că valoarea lui *p* trebuie să fie obținută printr-un apel al unei funcții standard de alocare, cum este de exemplu funcția *malloc*.

Se recomandă ca această funcție să fie apelată de îndată ce datele dintr-o zonă de memorie *heap* nu mai sînt necesare.

În felul acesta zona respectivă poate fi ulterior realocată. De asemenea, eliberarea sistematică a zonelor din memoria *heap* poate preveni fărîmîțarea excesivă a memoriei *heap*.

O altă funcție standard utilă pentru a alocă zone de memorie în memoria *heap* este funcția *calloc*. Ea are prototipul:

```
void *calloc(unsigned nrelem,unsigned dimelem);
```

Funcția alocă o zonă de memorie de *nrelem* \* *dimelem* octeți.

Ca și funcția *malloc*, funcția *calloc* returnează adresa de început a zonei de memorie alocată, adresa care reprezintă un pointer spre *void*.

În cazul în care nu se pot alocă *nrelem* \* *dimelem* octeți, funcția returnează valoarea zero.

Elementele zonei de memorie alocată prin *calloc* sînt inițializate cu valoarea zero.

Zona de memorie alocată cu ajutorul funcției *calloc* se eliberează folosind funcția *free* indicată mai sus.

Pentru programe relativ mici, pointerii se păstrează pe 16 biți. Aceasta înseamnă că ei pot păstra adrese de pînă la 64k. Despre un astfel de pointer se spune că este de tip *near*.

Programele mai complexe, care necesită adrese de peste 64k, utilizează pointeri alocați pe 32 de biți. Despre un astfel de pointer se spune că este de

tip *far*.

Biblioteca standard conține funcții pentru a aloca în memoria *heap* zone de memorie a căror adrese se reprezintă în memoria *heap* pe mai mult de 16 biți. De asemenea, dimensiunea unei astfel de zone de memorie poate depăși 64k. Astfel, pentru a aloca în memoria *heap* zone de memorie de adrese mai mari ca 64k, putem folosi funcțiile *farmalloc* și *farcalloc*, funcții analoge cu funcțiile *malloc* și respectiv *calloc*, indicate mai sus.

Funcția *farmalloc* are prototipul:

```
void far *farmalloc(unsigned long n);
```

Ea se utilizează la fel ca și funcția *malloc*. În acest caz funcția returnează un pointer de tip *far* (se alocă pe 32 de biți). De asemenea, la apelul funcției *farmalloc*, parametrul *n* poate depăși 64k = 65536.

Zona alocată cu ajutorul funcției *farmalloc* se eliberează apelînd funcția *farfree*, care este similară cu funcția *free*. Ea are prototipul:

```
void farfree(void far *p);
```

**Exerciții:**

8.15 Să se scrie o funcție care păstrează un șir de caractere într-o zonă de memorie alocată în memoria *heap*.

Funcția returnează adresa de început a zonei în care se păstrează șirul de caractere sau zero (pointerul nul), în cazul în care nu se poate rezerva zona respectivă în memoria *heap*.

## FUNCȚIA BVIII15

```
char *memsir(char *s)
/* pastreaza in memoria heap sirul de caractere spre care pointeaza s */
{
    char *p;

    if((p = (char *)malloc(strlen(s) + 1)) != 0) {
        /* p are ca valoare adresa de inceput a zonei de memorie rezervata in
        memoria heap si care are strlen(s) + 1 octeti */
        strcpy(p, s); /* copiaza sirul spre care pointeaza s, in zona
        spre care pointeaza p */
        return p;
    }
    else
```

```

        return 0; /* pointerul nul; nu s-a putut aloca zona necesara
                    pentru a pastra sirul spre care pointeaza s */
    }

```

#### Observații:

1. Funcția de față returnează un pointer spre caractere, deci *tip* din antetul ei este:

```
char *
```

2. Condiția din paranteza lui *if* se poate scrie și fără partea

```
!= 0
```

adică

```
if(p=(char *)malloc(strlen(s)+1))
```

Apelul:

```
strlen(s)
```

determină numărul de octeți necesari pentru a păstra caracterele proprii ale șirului spre care pointează *s*.

Numărul returnat de funcția *strlen* se mărește cu 1 pentru a aloca un octet pentru caracterul *NUL* care trebuie să termine orice șir de caractere.

3. Apelul:

```
strcpy(p,s);
```

copiază caracterele șirului, inclusiv caracterul *NUL*, spre care pointează *s*, în zona a cărei adresă de început este valoarea lui *p*.

- 8.16 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai mare.

Un astfel de program este descris în exercițiul 8.13. În programul respectiv, cuvântul maxim se păstrează în tabloul *cuvmax* alocat pe stivă. În programul de față se va păstra cuvântul maxim în memoria *heap*.

#### PROGRAMUL BVIII16

```

#include <stdio.h>
#include <string.h>
#include <alloc.h>

```

```

#include "bviii15.cpp"

#define MAX 100

main() /* citește o succesiune de cuvinte și-l afișează
        pe cel mai mare */
{
    char cuvrt[MAX+1];
    char *cuvmax = 0; /* pointează spre cuvântul maxim; inițial
                        are ca valoare pointerul nul */

    while(scanf("%100s", cuvrt) != EOF )
        if(cuvmax==0) /* prima citire */
            cuvmax = memsiv(cuvrt);
        else
            if(strcmp(cuvrt,cuvmax) > 0 ) {

/* cuvântul citit este mai mare decât cel din memoria heap și spre care
   pointează cuvmax */

/* se eliberează zona din memoria heap în care se păstrează cuvântul
   maxim întâlnit la citirile anterioare */
            free(cuvmax);

/* se alocă zona în memoria heap și se păstrează în ea cuvântul curent
   citit */
            cuvmax = memsiv(cuvrt);
        }
    printf("cel mai mare cuvânt este\n");
    printf("%s\n", cuvmax);
    free(cuvmax);
}

```

#### Observație:

În acest program nu s-a testat imposibilitatea păstrării șirului maxim în memoria *heap* deoarece acesta are cel mult 101 caractere. Ori memoria *heap* are alocat un spațiu mai mare.

8.17 Să se scrie o funcție care sortează în ordine crescătoare șiruri de caractere.

Funcția are doi parametri:

- tpointer* - Tablou unidimensional de tip *pointer* spre *char*.
- Fiecare element al tabloului este un *pointer* spre unul din cele *n* șiruri de caractere.
- *tpointer[i]* este *pointer* spre al *i + 1* - lea șir de caractere.
- n* - Întreg de tip *int* care are ca valoare numărul șirurilor de caractere care se sortează.

Funcția utilizează metoda bulelor. Pentru a ordona șirurile în ordine crescătoare se permută *pointerii* în locul șirurilor, ca în exercițiul 8.14.

La compararea șirurilor de caractere se ignoră diferența dintre literele mici și mari.

### **FUNCȚIA BVIII17**

```
void ordsircresc(char *tpointer[], int n)
/* sorteaza in ordine crescatoare cele n siruri de caractere spre care
   pointeaza elementele
   tpointer[0],tpointer[1],...,tpointer[n-1] */
{
    int ind,i;
    char *t;

    ind = 1;
    while( ind ) {
        ind = 0;
        for(i=0; i< n-1; i++)
            if(stricmp(tpointer[i],tpointer[i+1]) > 0){
                t = tpointer[i];
                tpointer[i] = tpointer[i+1];
                tpointer[i+1] = t;
                ind = 1;
            }
    }
}
```

8.18 Să se scrie un program care citește o succesiune de cuvinte și le afișează în ordine crescătoare. Se ignoră diferența dintre literele mici și mari.

Prin cuvânt se înțelege o succesiune de caractere care nu sînt albe. Cuvintele sînt separate prin caractere albe. Succesiunea de cuvinte se termină cu sfîrșitul de fișier.

Cuvintele citite se păstrează în memoria *heap*. Programul utilizează un tablou de *pointeri* ale cărui elemente *pointează* fiecare spre cite un cuvînt.

Se presupune că sînt cel mult 500 de cuvinte.

## PROGRAMUL BVIII18

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>

#include "bviii15.cpp"
#include "bviii17.cpp"

#define MAXCUV 500

main() /* citește o succesiune de cuvinte și le afișează în
       ordine crescătoare */
{
    char *tp[MAXCUV];
    int i,j;
    char t[255];

    /* se citesc cuvintele și se păstrează în memoria heap */
    for(i=0; scanf("%s",t) == 1; i++)
        if((tp[i] = memstr(t)) == 0 ) {
            printf("memorie insuficientă\n");
            exit(1);
        }

    /* se sortează cuvintele în ordine crescătoare */
    if( i )
        ordaircresc(tp,i);

    /* listarea cuvintelor după sortare */
    for(j=0; j < i; j++) {
        printf("%s\n", tp[j] );
        if((j+1)%23 == 0 ) {
            printf("actionati o tasta pentru a\
                    continua\n");
            getch();
        }
    } /* sfîrșit for listare */
}
```

### Observație:

Acest program are un avantaj față de cel din exercițiul 8.14, deoarece în cazul de față se alocă pentru fiecare cuvânt o zonă de memorie de lungime egală cu numărul de caractere ale cuvântului respectiv, plus 1 (pentru caracterul *NUL*). În programul din exercițiul 8.14, se alocă o zonă fixă de memorie de  $(30+1)*500$  octeți, care este de obicei prea mare, deoarece cuvintele au în medie mai puțin de 30 de caractere.

În general, alocarea dinamică în memoria *heap* permite alocări optime, față de cele făcute în mod obișnuit, folosind tablouri automate, globale sau statice.

## 8.9. Utilizarea tablourilor de pointeri la prelucrări de date de tip șir de caractere

Tablourile de pointeri pot fi utilizate pentru a prelucra succesiuni de șiruri de caractere. Utilizări de acest fel s-au văzut deja în exercițiile din paragraful precedent (vezi 8.18. și 8.14.).

Într-un program se întâlnesc adesea șiruri de caractere care se tratează într-un mod unitar. În acest caz este util să se definească un tablou de pointeri, fiecare element al său fiind un pointer spre un astfel de șir de caractere.

Astfel de tablouri pot fi utilizate pentru a păstra sau a avea acces la diferite denumiri, cuvinte cheie, mesaje de eroare etc.

În [2] se definesc astfel de tablouri pentru a păstra și utiliza denumirile lunilor calendaristice și cuvintele cheie ale limbajului C.

De exemplu, pentru a păstra denumirile lunilor calendaristice se poate utiliza un tablou de pointeri cu 13 elemente.

Numim *tpdl* acest tablou. Elementul *tpdl[1]* este pointer spre șirul de caractere "ianuarie", *tpdl[2]* este pointer spre șirul de caractere "februarie" etc.

Ultimul element, *tpdl[12]* este pointer spre șirul de caractere "decembrie".

Primul element al tabloului, nu pointează spre denumirea nici unei luni. De aceea, acest element poate avea ca valoare pointerul nul. O altă posibilitate este ca *tpdl[0]* să poarte spre un text de eroare, de exemplu "luna ilegală".

Dacă *p* este un pointer spre un șir de caractere, atunci poate fi definit sau



declarat ca pointer spre un șir de caractere printr-o construcție de forma:

```
char *p = sir;
```

Șirul de caractere *sir* se păstrează într-o zonă de memorie, iar lui *p* i se atribuie adresa de început a zonei respective.

În mod analog se pot inițializa elementele tabloului *tpdl*, folosind o definiție sau o declarație de forma:

```
char *tpdl[] = {  
    "luna ilegala",  
    "ianuarie",  
    "februarie",  
    "martie",  
    "aprilie",  
    "mai",  
    "iunie",  
    "iulie",  
    "august",  
    "septembrie",  
    "octombrie",  
    "noiembrie",  
    "decembrie"  
};
```

Prin această construcție, elementul *tpdl[i]*, pentru  $1 \leq i \leq 12$ , este pointer spre șirul de caractere care denumește luna calendaristică a *i* - a din an. De exemplu, apelul:

```
printf("%s\n",tpdl[1]);
```

afișează textul

```
ianuarie
```

**Exerciții:**

8.19 Să se scrie o funcție care are ca parametru un întreg *n* și returnează un pointer spre denumirea lunii calendaristice a *n* - a pentru  $1 \leq n \leq 12$  sau spre textul "luna ilegala" pentru "*n*" în afara acestui interval.

Fie *denluna* numele acestei funcții. Ea returnează un pointer spre *char*, deci returnează o dată de tip *char \**.

Deci antetul ei este:

```
char *denluna(int n)
```

## **FUNCȚIA BVIII19**

```
char *denluna(int n)
/* returneaza pointerul spre denumirea lunii */
{
    static char *tpdl[] = {
        "luna ilegala",
        "ianuarie",
        "februarie",
        "martie",
        "aprilie",
        "mai",
        "iunie",
        "iulie",
        "august",
        "septembrie",
        "octombrie",
        "noiembrie",
        "decembrie"
    };

    return n < 1 || n > 12 ? tpdl[0]: tpdl[n];
}
```

8.20 Să se scrie un program care citește o dată calendaristică scrisă sub forma:

zzllaaaa

o validează și o afișează sub forma:

zz luna aaaa

unde:

zz	- Reprezintă ziua pe două cifre.
ll	- Reprezintă numărul lunii pe două cifre.
aaaa	- Reprezintă anul pe 4 cifre.
luna	- Reprezintă denumirea lunii calendaristice.

Data citită se validează utilizind funcția *v\_calend* definită în exercițiul 6.5.

## **PROGRAMUL BVIII20**

```
#include <stdio.h>
```

```

#include <stdlib.h>

#include "bvi5.cpp"
#include "bviii19.cpp"

int nrzile[] = { 0,31,28,31,30,31,30,31,
                31,30,31,30,31 };

main() /* - citește o data calendaristică sub forma:
        zzllaaaa
        - o validează și o afișează sub forma:
        zz luna aaaa */
{
    int zz,ll,aaaa;

    if(scanf("%2d %2d %4d", &zz, &ll, &aaaa) != 3 ) {
        printf("nu s-a tastat un întreg de 8\
        cifre\n");
        exit(1);
    }
    if(v_calend(zz,ll,aaaa) == 0 ){
        printf("data calendaristică eronată\n");
        exit(1);
    }
    printf("%02d %s %4d\n", zz, denluna(ll),aaaa);
}

```

8.21 Să se scrie o funcție care din denumirea lunii calendaristice determină numărul ei.

Această funcție este inversa funcției *denluna* definită în exercițiul 8.19.

Funcția are ca parametru un pointer spre denumirea lunii calendaristice. Folosind acest pointer, denumirea lunii se caută cu ajutorul funcției *strcmp* pentru a stabili coincidența ei cu unul din șirurile de caractere spre care poartă elementele tabloului *tpdl* (vezi exercițiul 9.19.). În cazul în care denumirea este eronată funcția returnează valoarea zero.

## FUNCȚIA BVIII21

```

int    nrluna(char    *denl)
/* returnează numărul lunii din denumirea ei */
{

```

```

static char *tpdl[] = {
    "",
    "ianuarie",
    "februarie",
    "martie",
    "aprilie",
    "mai",
    "iunie",
    "iulie",
    "august",
    "septembrie",
    "octombrie",
    "noiembrie",
    "decembrie"
};
int i;

for( i=1; i <= 12; i++)
    if(strcmp(denl,tpdl[i]) == 0 )
        return i;
return 0;
}

```

8.22 Să se scrie un program care citește o dată calendaristică tastată sub forma:

zz luna aaaa

și o afișează sub forma:

zz/ll/aaaa

dacă este validă.

În aceste formate s-au utilizat notațiile:

zz	- Ziua pe 1 - 2 cifre.
ll	- Luna pe 1 - 2 cifre.
aaaa	- Anul pe 4 cifre.
luna	- Denumirea lunii calendaristice.

## PROGRAMUL IVV722

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

#include "bvi5.cpp"
#include "bviii21.cpp"

int nrzile[] = { 0,31,28,31,30,31,30,31,
                31,30,31,30,31 };

main() /*- citeste o data calendaristica sub forma:
        zz luna aaaa
        - o valideaza si o afiseaza sub forma:
        zz/ll/aaaa */
{
    int zz,ll,aaaa;
    char dl[11]; /* pastreaza denumirea lunii calendaristice */

    if(scanf("%2d %10s %4d", &zz, dl,&aaaa) != 3 ) {
        printf("data calendaristica eronata\n");
        exit(1);
    }

    /* determina numarul lunii calendaristice */
    if((ll = nrluna(dl)) == 0 ) {
        printf("luna eronata\n");
        exit(1);
    }
    if(v_calend(zz,ll,aaaa) == 0 ){
        printf("data calendaristica eronata\n");
        exit(1);
    }
    printf("%d/%d/%d\n", zz, ll, aaaa);
}

```

## 8.10. Tratarea parametrilor din linia de comandă

În linia de comandă folosită la apelul execuției unui program se pot utiliza diferiți parametri (argumente).

În cazul utilizării mediului integrat de dezvoltare Turbo C, acești parametri se definesc utilizând submeniul *Arguments* al meniului *Options*. Dacă se utilizează mediul integrat de dezvoltare *Borland C++*, atunci submeniul *Arguments* se selectează din meniul *Run*.

În toate cazurile argumentele sînt succesiuni de caractere separate prin spațiu. Ele se tastează în linia de comandă prin care se activează imaginea executabilă (fișierul cu extensia .exe) a programului.

Dacă se utilizează mediile integrate de dezvoltare Turbo C și Borland C++, atunci se procedează astfel:

#### mediul Turbo C

Se selectează meniul

*Option:*

<Alt>-O

Se selectează submeniul:

*Arguments:*

A

#### mediul Borland C++

Se selectează meniul

*Run:*

<Alt>-R

Se selectează submeniul:

*Arguments:*

A

După selectarea submeniului *Arguments* se afișează o fereastră în care se pot tasta argumentele separate prin spațiu. După ultimul argument se acționează tasta Enter. Apoi se poate continua cu lansarea automată a fazelor: compilare, link-editare și execuție. Se tastează:

<Ctrl>-F9

Pentru a utiliza aceste argumente se vor folosi parametrii *argc* și *argv* ai funcției principale. În acest caz funcția principală are antetul

```
main(int argc, char *argv[])
```

Parametrul *argc* are ca valoare numărul argumentelor din linia de comandă mărit cu unu. Parametrul *argv* este un tablou de pointeri spre zonele în care s-au păstrat argumentele definite ca mai sus.

Argumentele se păstrează sub formă de șiruri de caractere. Pointerul *argv[0]* este întotdeauna pointerul spre calea fișierului cu imaginea executabilă a programului. În continuare *argv[1]* este pointerul spre primul argument tastat, *argv[2]* este pointerul spre al doilea argument și așa mai departe. În general *argv[i]* este pointer spre al *i*-lea argument.

#### Exemplu:

Presupunem că la lansarea programului s-au furnizat argumentele:

1 OCTOMBRIE 1993

În cazul acesta parametrii funcției principale au valorile:

- |                 |   |
|-----------------|---|
| <i>argc</i> = 4 | - (3 argumente plus specificatorul fișierului cu imaginea executabilă a programului).                           |
| <i>argv[0]</i>  | - Pointer spre specificatorul fișierului cu imaginea executabilă a programului (cale, numele și extensia .EXE). |
| <i>argv[1]</i>  | - Pointer spre șirul "1".   |

`argv[2]`            - Pointer spre șirul "OCTOMBRIE".  
`argv[3]`            - Pointer spre șirul "1993".

### Exerciții:

8.23 Să se scrie un program care afișează parametri din linia de comandă.

### PROGRAMUL BVIII23

```
#include <stdio.h>
main( int argc, char *argv[])
/* afiseaza parametri din linia de comanda */
{
    int i;

    for( i=0; i < argc; i++)
        printf("%s\n",argv[i]);
}
```

8.24 Să se scrie un program care afișează data calendaristică preluată din linia de comandă sub forma:

`zz/ll/aaaa`

unde:

`zz`                    - Ziua pe 1 - 2 cifre.  
`ll`                    - Numărul lunii calendaristice pe 1 - 2 cifre.  
`aaaa`                  - Anul pe 4 cifre.

Programul apelează funcția `v_calend` pentru a valida data calendaristică preluată din linia de comandă. Întrucât ea se tastează sub forma indicată mai sus, acesta reprezintă un singur argument.

`argv[1]` este pointerul spre zona în care se păstrează data calendaristică respectivă.

### PROGRAMUL BVIII24

```
#include <stdio.h>
#include <stdlib.h>

#include "bvi5.cpp"
```

```

int nrzile[] = { 0,31,28,31,30,31,30,31,
                 31,30,31,30,31 };

main(int argc, char *argv[])
/* valideaza si afiseaza data calendaristica preluata din
   linia de comanda */
{
    int zz,ll,aaaa;
    char a,b;

    if(argc < 2 ) {
        printf("lipseste data calendaristica in linia\
              de comanda\n");
        exit(1);
    }
    if(argc > 2 ) {
        printf("mai mult de un argument in linia de\
              comanda\n");
        exit(1);
    }
    if(sscanf(argv[1],"%2d %c %2d %c %4d",&zz,&a,
               &ll,&b, &aaaa) != 5) {
        printf("argumentul nu are formatul\
              zz/ll/aaaa\n");
        exit(1);
    }
    if( a != '/' ) {
        printf("dupa zi nu urmeaza caracterul /\n");
        exit(1);
    }
    if(b != '/' ) {
        printf("dupa luna nu urmeaza caracterul /\n");
        exit(1);
    }
    if(v_calend(zz,ll,aaaa) == 0 ) {
        printf("data calendaristica eronata\n");
        exit(1);
    }
    printf("%d/%d/%d\n", zz, ll,aaaa);
}

```



## 8.11. Pointeri spre funcții

*Numele unei funcții este un pointer spre funcția respectivă. De aceea, numele unei funcții poate fi folosit ca parametru efectiv la apelul unei funcții. În felul acesta, o funcție poate transfera funcției apelate un pointer spre o funcție. Funcția apelată, la rindul ei, poate apela funcția care i-a fost transferată în acest fel.*

Fie, de exemplu, *f* o funcție care are ca parametru o altă funcție. Apelul:

*f(g);*

unde *g* este numele unei funcții, transferă funcției *f* pointerul spre funcția *g*.

Înainte de apel, ambele funcții (*f* și *g*) trebuie să fie definite sau să li se indice prototipurile.

Fie *g* de prototip:

*tpg g(listag);*

unde prin *tpg* s-a notat tipul valorii returnate de *g* sau *void* în cazul în care funcția *g* nu returnează o valoare, iar *listag* este lista cu tipurile parametrilor funcției *g* sau cuvântul *void* dacă *g* nu are parametri.

Să definim prototipul funcției *f*. Așa cum s-a indicat mai sus, ea are un parametru care este un pointer spre o funcție care are un prototip de forma prototipului lui *g* (numai numele poate diferi).

O construcție de forma:

*tip \*nume*

definește pe *nume* ca pointer spre tipul *tip*.

O construcție de forma:

*tip \*nume(lista)*

definește pe *nume* ca funcție care returnează un pointer spre *tip*.

Prezența parantezelor conduce la faptul că *nume* este o funcție și nu un pointer. Aceasta deoarece parantezele rotunde reprezintă operatori mai prioritari decât operatorul unar \*. Pentru ca *nume* să fie pointer, este nevoie ca operatorul unar \* să se aplice prioritar față de parantezele care indică prezența unei funcții. Aceasta se obține folosind o construcție de forma:

*tip (\*nume)(lista)*

În cazul de față *nume* este un pointer spre o funcție. Valoarea returnată de această funcție are tipul *tip*, iar *lista* definește tipurile parametrilor

acestei funcții.

În cazul funcției  $f$  de mai sus, parametrul ei formal se definește astfel:

```
tipg (*nume_par_formal)(listag)
```

Amintind faptul că într-un prototip se pot omite numele parametrilor formali, rezultă că prototipul funcției  $f$  poate fi scris astfel:

```
tipf f(tipg (*))(listag);
```

unde *tipf* este tipul valorii returnate de funcția  $f$  sau *void* în cazul în care  $f$  nu returnează o valoare.

Antetul funcției  $f$  va fi:

```
tipf f(tipg (*p))(listag)
```

unde *listag* definește tipurile parametrilor funcțiilor care se transferă la apelurile lui  $f$  prin pointerii spre ele

**Exemple:**

1. Funcția  $g$  are prototipul:

```
int g(void);
```

Funcția  $f$  care are la apel ca parametru efectiv pe  $g$  și returnează o valoare de tip *double*, are prototipul:

```
double f(int (*)(void));
```

2. Funcția  $h$  are prototipul:

```
double h(int, float);
```

Funcția  $f$  care are la apel ca parametru efectiv pe  $h$  și care returnează o valoare de tip *int*, are prototipul:

```
int f(double (*)(int, float));
```

Fie funcția  $f$  care are antetul:

```
tipf f(tip (*p))(lista)
```

Deci, parametrul ei  $p$  este un pointer spre o funcție care returnează o valoare de tipul *np*, iar tipurile parametrilor acestei funcții sînt definiți de *lista*.

La un apel de forma:

```
f(g);
```

lui  $p$  i se atribuie ca valoare pointerul spre funcția  $g$ . Se pune problema de a apela funcția  $g$  în corpul funcției  $f$ , folosind parametrul formal  $p$ . În acest

caz, apelul obișnuit al lui *g*:

*g*(...);

sau

*x*=*g*(...);

se schimbă înlocuind numele *g* (necunoscut în momentul definirii funcției *f*) prin *\*p*.

Deci în corpul funcției *f*, vom utiliza apeluri de forma:

(*\*p*)(...);

sau

*x*=(*\*p*)(...);

pentru a apela funcția care s-a transferat prin parametru.

### Exerciții:

8.25 Să se scrie o funcție care calculează și returnează valoarea aproximativă a integralei definite dintr-o funcție *f*(*x*), folosind metoda trapezului.

Să presupunem că integrala se calculează de la *a* la *b*. Atunci o valoare aproximativă a integralei se determină cu ajutorul formulei de mai jos:

$$In = h * ((f(a) + f(b))/2 + f(a+h) + f(a+2*h) + f(a+3*h) + ... f(a+(n-1)*h))$$

unde  $h = (b-a)/n$ .

Funcția de față calculează partea dreaptă a acestei relații. Ea are următorii parametri:

- |          |   |
|----------|---|
| <i>a</i> | - Limita inferioară - număr de tip <i>double</i> .  |
| <i>b</i> | - Limita superioară - număr de tip <i>double</i> .  |
| <i>n</i> | - Numărul de subintervale în care s-a împărțit intervalul [ <i>a</i> , <i>b</i> ] - întreg de tip <i>int</i> .  |
| <i>p</i> | - Pointerul spre funcția <i>f</i> ( <i>x</i> ) din care se calculează integrala - aceasta are un parametru de tip <i>double</i> și returnează o valoare de tip <i>double</i> ( <i>double f(double x)</i> ). |

### FUNCȚIA BVIII25

```
double itrapez(double a, double b,  
               int n, double(*p)(double))
```

*/\** - calculeaza prin metoda trapezului, integrala definita de la a la b din  
funcția spre care poanteaza *p*;

- *n* - este numărul subintervalelor în care s-a împartit intervalul [*a*,*b*]*\*/*

```

{
    double h,s;
    int i;

    h = (b - a) / n;
    s = 0.0;
    for(i=1; i < n; i++)
        s += (*p)(a+i*h);
    s += ((*p)(a) + (*p)(b)) / 2;
    s *= h;
    return s;
}

```

8.26 Să se scrie un program care calculează integrala definită din funcția:

$$f(x) = \sin(x^*x)$$

de la 0 la 1, folosind metoda trapezului. Se cere o eroare mai mică decît  $1e-8$ .

Pentru a obține precizia dorită vom proceda ca mai jos:

1. Se alege valoarea inițială pentru  $n$ , de exemplu 10.
2. Se calculează  $ln$ .
3. Se calculează  $l2n$  ( $n$  este dublat).
4. Dacă  $abs(l2n-ln) < 1e-8$ , atunci  $l2n$  este rezultatul și procesul de calcul se întrerupe.

Altfel se dublează  $n$ , se pune  $ln=l2n$  și se continuă cu pasul 3 de mai sus.

## PROGRAMUL BVIII26

```

#include <stdio.h>
#include <math.h>

#include "bviii25.cpp"

#define A 0.0
#define B 1.0
#define N 10
#define EPS 1e-8

double sinxp(double); /* prototip */

```

```

main() /* calculeaza integrala definita din functia
        sinxp=sin(x*x)
        in intervalul [A,B] cu o eroare mai mica decat EPS */
{
    int n = N;
    double in,i2n,vabs;

    in = itrapez(A,B,n,sinxp);
    do {
        n = 2*n;
        i2n = itrapez(A,B,n,sinxp);
        if((vabs = in - i2n) < 0 )
            vabs = -vabs;
        in = i2n;
    } while(vabs >= EPS );
    printf("integrala din sin(x*x)=%.10g\n",i2n);
    printf("numarul de subintervale n=%d\n", n);
} /* sfirsit main */

double sinxp(double x)
{
    return sin(x*x);
}

```

#### Observație:

Valoarea aproximativă a integralei din  $\sin(x^2)$  în intervalul  $[0,1]$  este 0.3102683026. Ea se obține pentru  $n=10240$  în aproximativ 40 de secunde, la un calculator fără coprocessor.

8.27 Să se scrie o funcție care determină rădăcina ecuației:

$$(1) f(x)=0$$

din intervalul  $[a,b]$ , cu o eroare mai mică decât  $\text{EPS}=1e-8$ , știind că ecuația are o singură rădăcină în intervalul respectiv și că  $f(x)$  este o funcție continuă pe același interval.

Din condițiile enunțate mai sus rezultă că:

$$f(a) \cdot f(b) \leq 0$$

sau

$$f(a) \cdot f(b) < 0$$

dacă nici una din limitele intervalului  $[a,b]$  nu sînt rădăcini ale ecuației (1).

O metodă simplă de calcul a rădăcinii ecuației (1) este prin metoda înjumătățirii. Ea constă în următoarele:

1.  $c = (a+b)/2$ .
2. Dacă  $f(c)=0$ , atunci  $c$  este soluția căutată și se întrerupe procesul de calcul.
3. Dacă  $f(a)*f(c) < 0$ , atunci se pune  $b=c$ , altfel  $a=c$ .
4. Dacă  $b-a < \text{EPS}$ , atunci procesul de calcul se întrerupe și rădăcina se ia media dintre  $a$  și  $b$ .

Altfel algoritmul se reia de la pasul 1.

## FUNCȚIA BVIII27

```
#define EPS 1e-8
```

```
double radec(double a, double b,  
             double (*p)(double))
```

/\* - calculeaza si returneaza solutia ecuatiei  $f(x) = 0$  din intervalul  $[a,b]$ ;

- ecuatia are o singura solutie in intervalul  $[a,b]$ ;

- p - este pointerul spre functia  $f(x)$  de prototip:

```
double f(double). */
```

```
{
```

```
double c,d;
```

```
if((*p)(a) == 0 )
```

```
    return a; /* a este radacina */
```

```
if((*p)(b) == 0 )
```

```
    return b; /* b este radacina */
```

```
if((*p)(a) * (*p)(b) > 0 ) {
```

```
    printf("ecuatia sau nu are radacina in\
```

```
        [a,b]\n");
```

```
    printf("sau are radacini multiple in\
```

```
        [a,b]\n");
```

```
    exit(1);
```

```
}
```

```
do {
```

```
    c = (a + b )/2;
```

```
    if((d = (*p)(c)) == 0 )
```

```
        return c; /* c este radacina */
```

```
    if((*p)(a)*d < 0 )
```

```
        b = c;
```

```

        else
            a = c;
    } while( b-a >= EPS );
    return (a+b)/2;
}

```

8.28 Să se scrie un program care calculează și afișează rădăcina ecuației:

$$x - \sin(x+1) = 0$$

din intervalul  $[0.5, 1]$ , cu o eroare mai mică decât  $1e-8$ .

### PROGRAMUL BVIII28

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "bviii27.cpp"

#define A 0.5
#define B 1.0

double f(double x) /* returneaza valoarea functiei x - sin(x+1) */
{
    return x - sin( x+1);
}

main () /* calculeaza si afiseaza radacina ecuatiei:
        x - sin(x+1) = 0
        din intervalul [0.5,1], cu o eroare mai mica decit 1.e-8 */
{
    printf("radacina ecuatiei: \n");
    printf("x - sin(x+1) = 0 este: %.10g\n",
        radec(A,B,f));
}

```

## 9. RECURSIVITATE

Spunem despre o funcție că este *recursivă* dacă ea se autoapelează. Ea se poate autoapela fie *direct*, fie *indirect* prin apelul altor funcții.

În limbajele C și C++ se pot defini funcții recursive fără a fi nevoie de a specifica acest lucru.

La apelurile recursive ale unei funcții aceasta este reapelată înainte de a se reveni din ea. La fiecare reapel a funcției, *parametrii* și *variabilele automate* ale ei se alocă pe stivă într-o zonă independentă. De aceea, aceste date au valori distincte la fiecare reapelare. Nu același lucru se întâmplă cu *variabilele statice*. Ele ocupă tot timpul aceeași zonă de memorie și deci ele își păstrează valoarea la un reapel.

Orice apel al unei funcții conduce la o revenire din funcția respectivă la punctul următor celui din care s-a făcut apelul.

Amintim că la revenirea dintr-o funcție se procedează la curățirea stivei, adică stiva se reface la starea ei dinaintea apelului. De aceea, orice reapel al unui apel recursiv va conduce și el la curățirea stivei, la o revenire din funcție și deci parametri și variabilele locale vor reveni la valorile lor dinaintea reapelului respectiv. Numai variabilele statice rămân neafectate la o astfel de revenire.

Funcțiile recursive se utilizează la definirea proceselor recursive de calcul.

Un proces de calcul se spune că este *recursiv*, dacă el are o parte care se definește prin el însuși. Un exemplu simplu de astfel de funcție este funcția de calcul al *factorialului*. Într-adevăr funcția *fact(n)* de calcul al factorialului se poate defini astfel:

$\text{fact}(n) = 1$ , dacă  $n = 0$ ;

altfel

$\text{fact}(n) = n * \text{fact}(n-1)$ .

Alternativa:

$\text{fact}(n) = n * \text{fact}(n-1)$

definește funcția *fact(n)* prin ea însăși.

Un proces recursiv trebuie totdeauna să conțină o parte care nu se definește prin el însuși. În exemplul de mai sus, alternativa:

$\text{fact}(n) = 1$ , dacă  $n = 0$



este partea care este definită direct.

Definiția de mai sus se transcrie imediat în limbajul C:

```
double fact(int n) /* calculează n! */
{
    if(n==0)
        return 1.0;
    else
        return n*fact(n-1);
}
```

Să urmărim execuția acestei funcții pentru  $n=3$ .

La apelul din funcția principală se construiește o zonă pe stivă în care se păstrează adresa de revenire în funcția principală, după apel, precum și valoarea lui  $n$ , ca în figura de mai jos.

**Stiva programului**

rev	adr1
n	3

Adresa de revenire în funcția principală  
care a făcut apelul

a

Deoarece  $n > 0$ , se execută alternativa *else*. Aceasta conține expresia:

(1)  $n*fact(n-1)$

care autoapelează (direct) funcția *fact*. Reapelarea funcției se realizează cu valoarea  $n-1 = 3-1 = 2$ . Prin aceasta se construiește o zonă nouă pe stivă, care conține revenirea la evaluarea expresiei de mai sus, precum și valoarea nouă a parametrului  $n$ , adică valoarea 2. Se obține starea din figura de mai jos.

**Stiva programului**

rev	adr2
n	2
rev	adr1
n	3

Adresa de revenire la evaluarea expresiei (1)

b

La noua reapelare a funcției *fact*,  $n=2 > 0$ , deci din nou se execută alternativa *else*.

Evaluarea expresiei (1) conduce la o reapelare cu valoarea  $n-1 = 2-1 = 1$ . Se obține configurația:

**Stiva programului**

rev	adr2
n	1
rev	adr2
n	2
rev	adr1
n	3

Adresa de revenire la evaluarea expresiei (1)

c

Din nou  $n = 1 > 0$  și deci se face o nouă reapelare și se obține configurația:

**Stiva programului**

rev	adr2
n	0
rev	adr2
n	1
rev	adr2
n	2
rev	adr1
n	3

Adresa de revenire la evaluarea expresiei (1)

d

În acest moment  $n=0$ , deci se revine din funcție cu valoarea 1. Se curăță stiva și se revine la configurația din figura c. Adresa de revenire permite continuarea evaluării expresiei (1). În acest moment  $n$  are valoarea 1 și cum s-a revenit tot cu valoarea 1, se realizează produsul

$$1 \cdot 1 = 1$$

După evaluarea expresiei (1) se realizează o nouă revenire tot cu valoarea 1. După curățirea stivei se ajunge la configurația din figura b. În acest moment  $n$  are valoarea 2. Evaluind expresia (1), se realizează produsul

$$2 \cdot 1 = 2.$$

Apoi se revine din nou din funcție cu valoarea 2. După curățirea stivei se ajunge la configurația din figura a.

În acest moment  $n$  are valoarea 3. Se evaluează expresia (1) și se obține:

$$3 \cdot 2 = 6$$

Se revine din funcție cu valoarea 6 ( $3!$ ) și conform adresei de revenire, se revine în funcția principală din care s-a apelat funcția *fact*.

În general, o funcție recursivă se poate realiza și nerecursiv, adică fără să se autoapeleze. De obicei, recursivitatea nu conduce la economie de memorie și nici la execuția mai rapidă a programelor. Ea permite însă o descriere mai compactă și mai clară a funcțiilor care exprimă procese de calcul recursive. Cu toate acestea, există cazuri când funcțiile recursive, deși au exprimări clare, ele nu sunt recomandate deoarece implică consum mare de timp și memorie, în comparație cu varianta nerecursivă a aceluiași proces de calcul. De exemplu, algoritmul de generare a permutărilor de  $n$  obiecte poate fi descris atât recursiv, cât și nerecursiv. O variantă recursivă simplă generează permutările de  $n$  obiecte inserind un obiect nou ( $n$ ) în toate pozițiile posibile în permutările de  $n-1$  obiecte. Acest algoritm, deși este mai simplu decât varianta nerecursivă, el nu este practic, deoarece generarea permutărilor de  $n$  obiecte presupune existența deja în memorie a permutărilor de  $n-1$  obiecte, ori  $n!$  crește foarte rapid, ceea ce conduce la un necesar mare de memorie chiar și pentru  $n$  mic. Se pot indica și alte funcții recursive inefficiente față de variantele lor nerecursive. De exemplu, în multe lucrări se prezintă funcția recursivă pentru calculul numerelor lui Fibonacci și se arată ineficiența ei față de varianta nerecursivă.

De aceea, rezolvările cu ajutorul funcțiilor recursive urmează să se adopte numai după un studiu atent al implicațiilor pe care le au acestea în privința necesarului de memorie și al timpului de execuție.

Există câteva tipuri de probleme care, de obicei, se rezolvă cu ajutorul funcțiilor recursive. Astfel, se recomandă să se utilizeze funcții recursive

pentru probleme a căror metode de rezolvare se pot defini recursiv.

În această clasă se includ metode de divizare, metode de căutare cu revenire (*backtracking*) etc.

În toate cazurile trebuie însă să se aibă în vedere necesarul de memorie și a timpului de execuție în comparație cu variantele nerecursive.

**Exerciții:**

- 9.1 Să se scrie un program care citește pe  $n$  de tip *int*, calculează și afișează  $n!$  folosind funcția recursivă *fact* definită mai sus.

## PROGRAMUL BIX1

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 170

double fact(int n) /* calculeaza pe n! */
{
    if(n == 0 )
        return 1.0;
    else
        return n * fact(n-1);
}

main () /* citeste pe n, calculeaza si afiseaza pe n! */
{
    int n;
    char t[255];

    for( ; ; ) {
        printf("valoarea lui n: ");
        if(gets(t) == 0) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d",&n) == 1 && n >= 0 && n <=
MAX )
            break;
        printf("se cere un intreg in intervalul\
[0,%d]\n", MAX);
    }
}
```

```
    printf("n= %d\t n!= %g\n", n, fact(n));
}
```

- 9.2 Să se scrie o funcție recursivă care calculează numărul aranjamentelor de  $n$  obiecte luate câte  $k$  ( $n \geq k > 0$ ).

Dacă notăm cu  $A(n,k)$  numărul aranjamentelor de  $n$  obiecte luate câte  $k$ , atunci:

$$A(n,k) = n(n-1)(n-2) \dots (n-k+1) = n(n-1)(n-2) \dots ((n-1)-(k-1)+1) = nA(n-1,k-1);$$

$$A(n,1) = n.$$

Relația de mai sus, pentru calculul numărului aranjamentelor, este recursivă. Ea se transcrie imediat în limbajul C folosind o funcție recursivă.

## FUNCȚIA BIX2

```
double aranjamente(int n, int k)
/* returneaza numarul aranjamentelor de n obiecte luate cate k
   (n>=k>0) */
{
    if( k == 1 )
        return (double) n;
    else
        return n*aranjamente(n-1,k-1);
}
```

- 9.3 Să se scrie un program care citește doi întregi  $n$  și  $k$

( $1 \leq n \leq 170, 1 \leq k \leq n$ ),

calculează și afișează numărul aranjamentelor de  $n$  obiecte luate câte  $k$ .

Valorile lui  $n$  și  $k$  se citesc folosind funcția `pcu_int` lim definită în exercițiul BVIII3.CPP. Această funcție, la rîndul ei, apelează funcția `pcu_int` definită în exercițiul BVIII2.CPP.

## PROGRAMUL BIX3

```
#include <stdio.h>
#include <stdlib.h>

#include "bviii2.cpp"
#include "bviii3.cpp"
#include "bix2.cpp"
```

```
#define MAX 170
```

```
main() /* - citește întregii n și k (1 ≤ n ≤ 170; 1 ≤ k ≤ n),
        - calculează și afișează numărul aranjamentelor de
        n obiecte luate câte k */
```

```
{
    int n,k;
    char er[]="S-a tastat EOF\n";

    if(pcit_int_lim("n= ",1,170,&n) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("k= ",1,n,&k) == 0 ) {
        printf(er);
        exit(1);
    }
    printf("n=%d\tk=%d\tA(n,k)=%g\n",n,k,
        aranjamente(n,k));
}
```

- 9.4 Să se scrie o funcție care are ca parametri doi întregi  $m$  și  $n$  de tip *long* și care calculează și returnează cel mai mare divizor comun al lor.

Notăm cu:

$(m,n)$

cel mai mare divizor comun al numerelor  $m$  și  $n$ . Calculul celui mai mare divizor comun a două numere se poate realiza recursiv astfel:

$(m,n) = m$	dacă $n = 0$ ;
$(m,n) = n$	dacă $m = 0$ ;
$(m,n) = (n,m\%n)$	dacă atât $m$ , cât și $n$ sînt diferiți de zero și $m > n$ (prin $m\%n$ s-a notat restul împărțirii lui $m$ la $n$ ).

## FUNCȚIA BIX4

```
long cmmdc(long m, long n)
/* calculează și returnează pe cel mai mare divizor comun al
   numerelor m și n */
{
    if( m == 0 )
        return n;
```

```

else
    if( n == 0 )
        return m;
    else
        if( m > n )
            return cmmdc( n, m%n);
        else
            return cmmdc( m, n%m);
}

```

- 9.5 Să se scrie un program care citește doi întregi pozitivi  $m$  și  $n$ , de tip *long*, calculează și afișează pe cel mai mare divizor comun al lor.

### PROGRAMUL BIX5

```

#include <stdio.h>
#include <stdlib.h>

#include "bix4.cpp"

main() /* citește pe m și n de tip long, calculează și afișează (m,n) */
{
    long m,n;

    printf("valorile lui m și n: ");
    if(scanf("%ld %ld", &m, &n) != 2 || m <= 0
        || n <= 0 ) {
        printf("nu s-au tastat doi întregi\
pozitivi\n");
        exit(1);
    }
    printf("m=%ld\tn=%ld\t(m,n)=%ld\n",m,n,cmmdc(m,n))
}

```

- 9.6 Să se scrie un program pentru rezolvarea problemei *tururilor din Hanoi*. Această problemă se enunță ca mai jos.

Se consideră trei tije A, B, C și  $n$  discuri de diametre diferite. Tijele sunt fixate vertical pe o placă. Discurile au fiecare câte un orificiu în centru și ele pot fi aranjate pe fiecare din cele trei tije. Inițial toate discurile sunt puse pe tija A și ordonate în așa fel încât orice disc este așezat peste un disc de diametru mai mare decât al lui. Dacă numerotăm discurile în ordinea crescătoare a diametrelor lor, adică discul 1 are diametrul cel mai mic, apoi discul 2 are diametrul următor și așa mai departe, discul  $n$  având diametrul

cel mai mare, atunci discul 1 se află în vîrf, sub el discul 2 etc., iar la bază se află discul  $n$ .

Se cere să se mute discurile de pe tija A pe tija B, folosind tija C ca tijă de manevră. La fiecare mutare se mută un singur disc și anume, unul aflat în vîrfurile discurilor de pe o tijă se pune în vîrfurile discurilor de pe o altă tijă sau pe placă dacă tija respectivă nu conține discuri. De asemenea, nu se poate muta un disc de diametru mai mare peste unul de un diametru mai mic. Deci, la o mutare, peste discul  $k$  se poate așeza numai unul din discurile  $1, 2, \dots, k-1$ . În final discurile trebuie să fie așezate pe tija B în aceeași ordine în care s-au aflat pe tija A, adică în vîrf să fie discul 1, sub el discul 2 și așa mai departe, discul  $n$  aflîndu-se la bază.

Problema turnurilor din Hanoi pentru  $n$  discuri se rezolvă imediat dacă știm să o rezolvăm pentru  $n-1$  discuri. Într-adevăr, în ipoteza că știm să rezolvăm problema pentru  $n-1$  discuri, procedăm astfel:

1. Se deplasează discurile  $1, 2, \dots, n-1$  de pe tija A pe tija C.
2. Discul  $n$  se mută de pe tija A pe tija B.
3. Cele  $n-1$  discuri de pe tija C se deplasează pe tija B.

În felul acesta, pe tija B se află toate cele  $n$  discuri și în ordinea cerută, adică  $1, 2, \dots, n$ .

Deci rezolvarea problemei turnurilor din Hanoi cu  $n$  discuri s-a redus la rezolvarea aceleiași probleme, dar cu  $n-1$  discuri. În mod analog, problema turnurilor din Hanoi cu  $n-1$  discuri se poate rezolva dacă știm să rezolvăm problema respectivă pentru  $n-2$  discuri. În felul acesta, din aproape în aproape, ajungem la concluzia că problema turnurilor din Hanoi pentru  $n$  discuri se poate rezolva, dacă știm să o rezolvăm pentru  $n=1$ . Ori în acest caz problema este imediată: se mută discul de pe tija pe care se află (tija sursă), pe tija pe care dorim să se afle discul (tija destinație).

Cele trei tije se afla în una din următoarele stări posibile:

- tijă sursă* - Conține discurile care trebuiesc transferate pe altă tijă.
- tijă destinație* - Tijă pe care trebuie să se afle discurile transferate de pe tija sursă.
- tijă de manevră* - Tijă care se folosește pentru a stoca temporar discuri.

Rolul tijelor se schimbă în diferite etape ale procesului de deplasare a discurilor și de aceea, rolul fiecărei tije se definește cu ajutorul parametrilor

Mai jos definim o funcție recursivă pe care o numim *hanoi*. Ea are 4 parametri:

- $n$  - Numărul discurilor.



- a* - Tija sursă.
- b* - Tija destinație.
- c* - Tija de manevră.

. Apelul:

`hanoi(n,'A','B','C');`

înseamnă că se rezolvă problema pentru *n* discuri. Acestea se află pe tija A, tija B este tija destinație și tija C este tija de manevră.

Funcția *hanoi* afișează fiecare mutare, indicând numărul discului, tija de pe care se mută discul respectiv, precum și tija pe care se mută el.

După afișarea unei mutări, se apelează funcția *getch* pentru a bloca execuția programului. În felul acesta se poate analiza mutarea afișată de calculator.

Menționăm că problema de față poate fi rezolvată și nerecursiv. Varianta nerecursivă este mai complexă decât cea recursivă.

## PROGRAMUL BIX6

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void hanoi(int n, int a, int b, int c )
/* funcție recursivă pentru rezolvarea problemei turnurilor din Hanoi,
   n - numărul discurilor;
   a - tija sursă;
   b - tija destinație;
   c - tija de manevră. */
{
    if(n == 1 ) { /* n=1 */
        printf("se muta discul 1 de pe tija: %c pe\
               tija: %c\n",a,b);
        getch();
        return;
    }

    /* n > 1;
       se rezolvă problema pentru n-1 discuri;
       a - tija sursă; c - tija destinație; b - tija de manevră. */
    hanoi(n-1, a, c, b);
```

```

/* discul n de pe tija a se muta pe tija b */
printf("discul: %d de pe tija: %c se muta pe\
      tija: %c\n",n,a,b);
getch();

/* se rezolva problema pentru n-1 discuri;
   c - tija sursa; b - tija destinatie; a - tija de manevra. */
hanoi(n-1, c, b, a);
}
/* sfirsit hanoi */

main () /* citeste pe n si rezolva problema turnurilor din Hanoi
        pentru n discuri */
{
    int n;
    char t[255];

    for(;;) {
        printf("numarul discurilor= ");
        if(gets(t) == 0) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if (sscanf(t,"%d",&n) == 1 && n > 0 )
            break;
        printf("nu s-a tastat un intreg\
              pozitiv\n");
    }
    hanoi(n, 'A', 'B', 'C');
}

```

## 10. STRUCTURI ȘI TIPURI DEFINITE DE UTILIZATOR

Limbajele de programare oferă utilizatorului facilități de prelucrare atât a datelor *singulare* (izolate), cit și a celor *grupate*.

Datele se grupează pentru a forma mulțimi de elemente care să poată fi prelucrate atât element cu element, cit și global. De obicei, aceste grupe sînt mulțimi ordonate de date, adică datele unei astfel de grupe satisfac anumite relații.

Un mod simplu de grupare a datelor ne conduce la noțiunea de *tablou*.

*Tabloul* este o mulțime ordonată de date de un *același tip*, relația de ordine între elementele sale fiind definită cu ajutorul indicilor. *Numărul* indicilor definește *dimensiunea* tabloului. Tipul comun elementelor tabloului este și *tipul* tabloului.

Adesea este util să grupăm date într-un alt mod decît cel utilizat în cazul tablourilor. Astfel, se pot grupa date care nu neapărat sînt de același tip. Gruparea se face cu scopul de a prelucra datele respective nu numai separat, individual ci și global. Grupa și în acest caz este o mulțime ordonată. Referirea la elementele ei nu se mai face însă folosind indici, ci construcții de felul numelui. Componentele unei grupe pot fi ele însele grupe de alte date și așa mai departe. În felul acesta, datele sînt grupate pe *nivele*. Grupa care nu este componentă a unei alte grupe se spune că se află la nivelul *cel mai înalt*. Datele care nu mai sînt grupe de alte date se numesc *date elementare*. Ele se află la nivelele cele *mai inferioare*. În felul acesta datele sînt grupate potrivit unei ierarhii. Datele grupate, conform unei ierarhii, se numesc *structuri*.

Un exemplu simplu de structură este *data calendaristică*. Ea este o grupă de 3 date elementare: *zi, lună și an*.

Componentele *zi și an* sînt date de tip întreg. Componenta *lună* poate fi un șir de caractere, care să reprezinte luna prin denumirea ei: ianuarie, februarie etc. De aceea, componenta *lună* este un tablou de tip caracter care permite memorarea denumirii unei luni calendaristice.

### Observație:

Datele elementare ale unei structuri pot fi izolate sau tablouri.

Dacă în cazul tablourilor, tipul lui este tipul comun elementelor sale, în cazul structurilor fiecare structură reprezintă un *tip nou* de date. Un astfel

de tip se introduce printr-o *declarație de structură* și se spune că este un *tip definit de utilizator*.

În capitolul de față vom trece în revistă aspectele de bază cu privire la definirea și folosirea structurilor și a construcțiilor înrudite cu ele (reuniuni, câmpuri de biți etc.).

## 10.1. Declarația de structură

Pentru a declara o structură se pot folosi mai multe formate. Un format utilizat frecvent este:

```
struct nume {  
(1)   lista_de_declaratii  
} nume1,nume2,...,numen;
```

unde:

*nume,nume1, nume2,...,numen* - Sînt *nume* care pot și lipsi, dar nu toate decodată.

Dacă *nume* este absent, atunci cel puțin *nume1* trebuie să fie prezent.

Dacă lista *nume1,nume2,...,numen* este absentă, atunci trebuie ca *nume* să fie prezent.

Dacă *nume* este prezent, atunci el definește un *tip nou*, introdus prin declarația de structură respectivă. El poate fi utilizat în continuare pentru a defini date de acest tip, în mod asemănător cum definim date de tipuri predefinite.

În declarația de mai sus, *nume1,nume2,...,numen* sînt structuri de tipul *nume*.

În declarația de mai sus *numei* ( $i=1,2,...,n$ ) se poate înlocui cu:

*numei* [*lim1*] [*lim2*]...[*limk*]

și atunci *numei* este un tablou *k*-dimensional de elemente de tip *nume*.

Prin *lista\_de\_declaratii* înțelegem una sau mai multe declarații prin care se declară componentele structurii de tip *nume*.

O structură de tip *nume* poate fi declarată și ulterior, folosind formatul:

```
(2) struct nume nume_de_structura;
```

unde:

*nume\_de\_structura* - Este un *nume*.

Această declarație seamănă cu o declarație obișnuită în care se folosesc tipuri predefinite:

```
tip nume_data_izolata;
```

Deci, *tip* dintr-o declarație obișnuită se înlocuiește cu *struct nume*, unde *nume* este definit în prealabil printr-o declarație de structură de forma (1), în care *nume1, nume2, ..., numen* pot și lipsi.

Exemple: \_

1. Declaram *data\_nasterii* și *data\_angajarii* ca structuri de tip *data\_calendaristica*, care este compusă din cele trei date amintite mai sus: *zi, luna* și *an*:

```
struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
} data_nasterii, data_angajarii;
```

Declarația de față este de forma (1).

În cazul în care nu dorim să introducem tipul *data\_calendaristica*, se poate utiliza declarația:

```
struct {  
    int zi;  
    char luna[11];  
    int an;  
} data_nasterii, data_angajarii;
```

În sfârșit, este posibil să definim tipul utilizator *data\_calendaristica* și apoi să declarăm datele *data\_nasterii* și *data\_angajarii* folosind formatul (2).

```
struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
} ;  
  
struct data_calendaristica data_nasterii,  
    data_angajarii;
```

Prima declarație introduce tipul utilizator *data\_calendaristica*.

Declarația a doua definește datele *data\_nasterii* și *data\_angajarii* ca fiind structuri de tipul *data\_calendaristica*.

2. Fie tipul *data\_calendaristica* definit mai sus. Declarația:

```
struct data_calendaristica d[100];
```

definește pe *d* ca un tablou de 100 de elemente, fiecare element fiind de tipul *data\_calendaristica*.

3. Considerăm un model simplificat de date personale:

- nume\_prenume;
- adresa;
- localitatea\_nasterii;
- cod;
- data\_nasterii;
- data\_angajarii;
- sex;
- stare\_civila;
- numar\_copii.

Prin declarația de mai jos introducem tipul *date\_pers*:

```
struct date_pers {  
    char nume_prenume[100];  
    char adresa[100];  
    char localitatea_nasterii[100];  
    long cod;  
    struct data_calendaristica  
        data_nasterii,  
        data_angajarii;  
    char sex;  
    char starea_civila[15];  
    int numar_copii;  
};
```

La definirea tipului *date\_pers* s-a utilizat tipul *data\_calendaristica*. Acest lucru este posibil dacă tipul *data\_calendaristica* este în prealabil definit.

Fie declarația:

```
struct date_pers unangajat, nangajati[1000];
```

Prin această declarație se declară structura *unangajat* și tabloul de structuri *nangajati*. Ambele au tipul *date\_pers*.

4. În acest exemplu se introduce tipul *complex*:

```

    struct complex {
        double real;
        double imag;
    };

```

Datele declarate cu ajutorul declarației:

```

struct complex z,tz[100];

```

sînt date de tip *complex*. Atît partea reală, cit și partea imaginară sînt date de tip *double*.

5. Fie declarația:

```

    struct dmatp2 {
        double matrice[2][2];
    } ;
    struct dmatp2 a,b;

```

Variabilele *a* și *b* sînt fiecare cite o matrice pătratică de ordinul 2 a căror elemente sînt numere de tip *double*.

6. Fie declarația:

```

    struct elev {
        char *nume;
        char *prenume;
        long nr_matricol;
        struct data_calendaristica
            data_nasterii;
        char *adresa;
        char *locul_nasterii;
        int clasa;
        int note[15];
    } ;

```

Prin această declarație s-a introdus tipul *elev*. Declarația următoare definește date de tipul *elev*:

```

struct elev unelev,claselevi[25];

```

7. Ecranul pe care se afișează datele poate fi gestionat în mod *text* sau în mod *grafic*. În mod implicit, ecranul se gestionează în mod *text*. În acest mod, ecranul se compune, de obicei, din 25 linii a 80 de coloane.

Poziția unui punct pe ecran se definește prin 2 valori (coordonate):

*x* - numărul coloanei

și

y - numărul liniei.

Colțul din stînga sus are coordonatele:

x = 1;

y = 1.

Colțul din dreapta jos are coordonatele:

x = 80;

y = 25.

Pentru a controla afișarea datelor pe ecran, se pot utiliza diferite funcții standard de gestiune a ecranului. O parte dintre aceste funcții folosesc coordonatele x și y de felul celor indicate mai sus.

Adesea este util să definim tipul *punct* printr-o declarație de felul următor:

```
struct punct {  
    int x;  
    int y;  
};
```

Cu ajutorul acestui tip se pot defini poziții pe ecran:

```
struct punct poz;
```

Structurile sînt date care se pot alocă la fel ca și celelalte date (variabile simple sau tablouri). Astfel, o structură este *globală* dacă se declară în afara corpului oricărei funcții, este *automatică* dacă se declară în corpul unei funcții sau *statică* dacă declarația ei începe cu cuvîntul cheie *static*.

Datele elementare care sînt componente ale unei structuri se pot inițializa. În acest scop, într-o declarație sau definiție de structură, după numele structurii, se scrie caracterul = , iar după acesta construcțiile prin care se vor inițializa componentele elementare ale structurii, separate prin virgulă și incluse între acolade. Construcțiile utilizate la inițializare sînt expresii constante care corespund elementelor pe care le inițializează.

Exemple:

1.

```
struct data_calendaristica  
dc= { 1, "septembrie", 1994};
```

unde:

*data\_calendaristica* - Este tipul declarat în prealabil astfel:



```

    struct data_calendaristica {
        int zi;
        char luna[11];
        int an;
    };

```

2. Fie tipul *complex*:

```

    struct complex {
        double real;
        double imag;
    };

```

Declarația de mai jos definește variabila complexă *z*, care inițial are valoarea:

$1 + 2i$

```

    struct complex z = { 1.0, 2.0 };

```

Componentele elementare neinițializate ale unei structuri au valoarea inițială *zero* dacă structura este *globală* sau *stancă*.

În cazul structurilor *automatice*, elementele neinițializate au o valoare inițială *nedefinită*.

Operatorul unar adresă se poate aplica numelui unei structuri. Ca rezultat se obține adresa primei sale componente elementare. De asemenea, se pot defini pointeri spre tipuri utilizator.

**Exemplu:**

```

    struct complex {
        double real;
        double imag;
    };
    struct complex *p;
    struct complex z;
    ...
    p = &z;

```

## 10.2. Accesul la elementele unei structuri

Fie tipul utilizator *data\_calendaristica* definit în paragraful precedent:

```

    struct data_calendaristica {
        int zi;
        char luna[11];
    };

```

```
int an;  
};
```

și variabila *dc* de tip *data\_calendaristica*:

```
struct data_calendaristica dc;
```

Data *dc* are 3 componente:

*zi, luna și an* - Sînt date elementare.

Se pune problema accesului (referirii) la componentele structurii *dc*.

Referirea la componentele unei structuri se face utilizînd atît numele structurii cit și a componentei respective. Aceasta se realizează printr-o construcție de forma:

*nume.nume1*

unde:

*nume* - Este numele structurii.

*nume1* - Este numele componentei la care se dorește să se facă acces.

În exemplul de mai sus, vom avea acces la componenta *zi* a structurii *dc*, prin construcția:

*dc.zi*

În mod analog:

*dc.an*

ne asigură accesul la componenta *an* a structurii *dc*. Datele *dc.zi* și *dc.an* sînt de tip *int*.

Construcția *dc.luna* este un pointer spre caractere și permite accesul la componenta *luna* a structurii *dc*.

Punctul utilizat într-o construcție de forma:

(1)*nume.nume1*

se consideră un *operator* și el are prioritate *maximă*, la fel ca și operatorii paranteză.

O construcție de forma (1) se spune că este un *nume calificat*.

**Exemple:**

1.

```
struct data_calendaristica {
```

```

    int zi;
    char luna[11];
    int an;
};
struct data_calendaristica dc, d[10];
...
dc.zi = 1;
dc.an = 1995;
strcpy(dc.luna, "septembrie");
...
d[3].zi = dc.zi;
d[3].an = dc.an;
strcpy(d[3].luna, dc.luna);
...

```

Primele 3 instrucțiuni, atribuie structurii *dc* data calendaristică:

1 septembrie 1995

Ultimele 3, atribuie elementului *d[3]* valorile componentelor structurii *dc*, deci elementul respectiv păstrează aceeași dată calendaristică.

2.

```

struct data_calendaristica {
    int zi;
    char luna[11];
    int an;
};
struct date_pers {
    char nume[50];
    char prenume[50];
    long cod;
    struct data_calendaristica
        data_nasterii,
        data_angajarii;
};
struct date_pers angajat, sectie[100];
strcpy(angajat.nume, "Popescu");
strcpy(angajat.prenume, "Gheorghe");
angajat.cod = 123456789;

```

Pentru a defini data nașterii va trebui să folosim o dublă calificare:

```

angajat.data_nasterii.zi = 3;
strcpy(angajat.data_nasterii.luna, "august");

```

```
angajat.data_nasterii.an = 1970;
```

În mod analog se definește data angajării:

```
angajat.data_angajarii.zi = 15;  
strcpy(angajat.data_angajarii.luna, "septembrie");  
angajat.data_angajarii.an = 1993;
```

În limbajul C structurile nu se pot transfera prin parametri, ci numai pointerii spre ele. Așa de exemplu, structura *angajat* poate fi prelucrată de funcția *f*, dacă aceasta se apelează cu parametrul efectiv *&angajat* (adresa de început a zonei de memorie alocată structurii *angajat*):

```
f(&angajat);
```

Funcția *f* are ca parametru un pointer spre tipul structurii *angajat*, adică spre tipul: *date\_pers*.

Rezultă că funcția *f* are antetul:

```
void f(struct date_pers *p)
```

Într-adevăr, declarația lui *p* specifică faptul că el are ca valoare adresa de început a unei zone de memorie în care se păstrează date de tip *date\_pers*. Ori *&angajat* este chiar o astfel de adresă.

Problema care se pune în continuare este aceea a accesului la componentele structurii transferate prin adresa ei. În cazul de față se pune problema de a avea acces, în corpurile funcției *f*, la componentele structurii *angajat*: nume, prenume etc.

În corpul funcției *f* nu se pot utiliza numele calificate:

```
angajat.nume, angajat.prenume, angajat.data_nasterii.zi etc.
```

deoarece nu este cunoscut numele structurii *angajat*. În locul numelui structurii se cunoaște pointerul spre ea: *p*. De aceea, în construcțiile de mai sus nu avem decît să înlocuim numele *angajat* cu *\*p*. Se vor obține construcțiile:

```
(*p).nume;  
(*p).prenume;  
(*p).data_angajarii.zi  
etc.
```

Pentru a simplifica scrierea construcțiilor de acest fel s-a introdus o notație cu săgeată ( -> minus urmat de caracterul mai mare) și anume:

```
(*p). se înlocuiește cu p ->
```

În felul acesta, construcțiile de mai sus se scriu mai simplu:

p -> nume;  
p -> prenume;  
p -> data\_angajarii.zi etc.

Săgeata, ca și punctul, se consideră un operator de prioritate *maximă*, la fel ca și parantezele. Amintim că acești operatori se asociază de la stînga spre dreapta.

În concluzie, accesul la componentele unei structuri se realizează fie printr-o construcție de forma:

*nume.nume1*

fie prin:

*pointer -> nume1*

unde:

<i>nume</i>	- Este numele structurii.
<i>nume1</i>	- Este numele componentei.
<i>pointer</i>	- Este un pointer spre structură.

### 10.3. Asignări de nume pentru tipuri de date (declarații de tip)

Tipurile predefinite se identifică printr-un cuvînt cheie. În cazul tipurilor utilizator în locul cuvîntului cheie se utilizează construcția:

**struct nume**

unde:

*nume* - Este introdus printr-o declarație de forma:

```
struct nume {  
    ...  
};
```

În limbajul C se poate atribui un nume unui tip, indiferent că el este un tip predefinit sau unul utilizator.

În acest scop se utilizează o construcție de forma:

**(1) typedef tip nume\_tip;**

unde:

<i>tip</i>	- Este fie un tip predefinit, fie un tip utilizator.
<i>nume_tip</i>	- Este numele care se atribuie tipului definit de <i>tip</i> .

După ce s-a atribuit un nume unui tip, numele respectiv poate fi utilizat pentru a declara date de acel tip, exact la fel cum se utilizează în declarații cuvintele cheie ale tipurilor predefinite: *int*, *char*, *float* etc.

De obicei, numele atribuit unui tip se scrie cu litere mari. Construcția (1) de mai sus o vom considera ca fiind o declarație prin care se atribuie un nume unui tip sau mai scurt *declarație de tip*.

**Exemple:**

1. Fie declarația:

```
typedef int INTREG;
```

După această declarație, cuvântul INTREG se poate utiliza pentru a defini date de tip *int*. Deci declarația:

```
INTREG x;
```

este identică cu declarația:

```
int x;
```

2. Fie declarația:

```
typedef float REAL;
```

Datele:

```
REAL a,b;
```

sînt de tip *float*.

3.

```
typedef struct data_calendaristica {  
    int zi;  
    char luna[11];  
    int an;  
} DC;
```

Declarația:

```
DC data_nasterii, data_angajarii;
```

este identică cu declarația:

```
struct data_calendaristica data_nasterii, data_angajarii;
```

4.

```
typedef struct {  
    double real;
```

```

    double imag;
} COMPLEX;

```

În continuare se pot declara numere complexe prin declarații de forma:  
**COMPLEX z, tz[10];**

5.

```

typedef struct {
    int x;
    int y;
} PUNCT;
PUNCT a, b;

```

Variabilele *a* și *b* sînt structuri de tip **PUNCT**. Fiecare are o componentă *x* (abscisa) și una *y* (ordonata).

6. Un dreptunghi se poate defini prin două vîrfuri diametral opuse ale sale. De obicei, se consideră vîrful din colțul stînga sus și cel din dreapta jos.

Definim tipul utilizator **DREPTUNGHI**:

```

typedef struct {
    PUNCT stanga_sus;
    PUNCT dreapta_jos;
} DREPTUNGHI;

```

În continuare se pot defini date de tip **DREPTUNGHI**:

**DREPTUNGHI d, td[10];**

7. **typedef int \*PINT;**

**PINT** este numele atribuit pentru tipul pointer spre *int*.

Declarația:

**PINT p, t[100];**

este identică cu declarația:

**int \*p, \*t[100];**

Construcția:

*tip (\*p) (lista de tipuri)*

se poate folosi în antetul unei funcții și cu ajutorul ei, parametrul *p* se declară ca fiind pointer spre o funcție care returnează o valoare de tipul *tp*, iar *lista de tipuri* definește tipurile parametrilor funcției spre care poartea *p*. La tipul parametrului *p* de mai sus se poate atribui un nume printr-o construcție *typedef* de forma:

(2) `typedef tip (*nume_tip)(lista tipurilor);`

**Exemple:**

1. Fie funcția *f* de antet:

```
void f(int (*p)(float,char,long,int) )
```

Declarația:

```
typedef int (*PF) (float,char,long,int);
```

ne permite să rescriem mai simplu antetul funcției *f*:

```
void f( PF p )
```

2. Folosind declarația:

```
typedef double (* PDF) (double);
```

putem rescrie antetul funcției *itrapez* din exercițiul 8.25, în felul următor:

```
double itrapez(double a, double b, int n, PDF p)
```

Din cele de mai sus se observă că declarațiile *typedef* ne permit adesea să facem simplificări la scrierea antetelor și prototipurilor funcțiilor. Acestea devin substanțiale mai ales atunci când apar expresii complicate cu pointeri.

Sensul unei expresii cu pointeri se poate stabili dacă se exprimă prin cuvinte efectul fiecărui operator, ținând seama de regulile de prioritate și asociativitate.

În lucrarea [5] se dau exemple de expresii complexe cu pointeri și se stabilesc sensurile lor înlocuind prin cuvinte efectele operatorilor pe care-i conțin.

Un astfel de exemplu indicat în lucrarea [5] este următoarea declarație:

```
int *(*(*x)[6]) () ;
```

Sensul lui *x* se determină astfel:

*int \** definește tipul pointer spre *int*, deci:

- |    |                            |  |
|----|----------------------------|--|
| a. | <code>(*(*x)[6]) ()</code> | - Este pointer spre <i>int</i> .   |
| b. | <code>(*(*x)[6])</code>    | - Este o funcție și ținând seama de afirmația de la punctul a., rezultă că această funcție returnează un pointer spre <i>int</i> . |
| c. | <code>*(*x)[6]</code>      | - Are același sens cu cel de la punctul b.   |
| d. | <code>(*x)[6]</code>       | - Pointer spre construcția definită precedent, deci pointer spre o funcție care returnează un pointer                              |



spre întregi.

- e. `(*x)` - Tablou de 6 elemente, fiecare element este un pointer spre o funcție care returnează un pointer spre întregi.
- f. `*x` - Construcție identică cu cea de la punctul e.
- g. `x` - Este pointer spre un tablou de 6 elemente, fiecare element fiind un pointer spre o funcție care returnează un pointer spre *int*.

Un alt exemplu din aceeași lucrare este declarația:

```
char (*( *y())[])();
```

Această declarație se interpretează astfel:

- a. `((*y())[])()` - Definește o valoare de tip *char*.
- b. `((*y())[])` - Este o funcție care returnează o valoare de tip *char*.
- c. `((*y())[])` - Este aceeași construcție ca și cea de la punctul b.
- d. `(*y())[]` - Este un pointer spre o funcție care returnează o valoare de tip *char*.
- e. `(*y())` - Este un tablou de pointeri spre funcții care returnează o valoare de tip *char*.
- f. `*y()` - Este aceeași construcție ca și cea de la punctul e.
- g. `y()` - Este pointer spre un tablou de pointeri spre funcții care returnează o valoare de tip *char*.
- h. `y` - Este o funcție care returnează un pointer spre un tablou de pointeri spre funcții care returnează o valoare de tip *char*.

Declarațiile de acest gen pot fi simplificate folosind declarații de tip.

De exemplu, în lucrarea [5] se declară tipuri intermediare pentru a simplifica declarația:

```
int (*( *x)[6])();
```

analizată mai sus.

**Exerciții:**

10.1 Să se scrie o funcție care calculează și returnează modulul unui număr complex.

Dacă:

$$z = x + iy$$

atunci modulul numărului complex este rădăcina pătrată din:

$$x^2 + y^2$$

## FUNCȚIA BX1

```
double dmodul (COMPLEX *z)
/* calculeaza si returneaza modulul numarului complex z */
{
    return sqrt(z -> x * z -> x + z -> y * z -> y);
}
```

10.2 Să se scrie o funcție care calculează și returnează argumentul unui număr complex.

Dacă:

$$z = x + iy$$

atunci

$$\arg z$$

se calculează astfel:

- a. Dacă  $x = y = 0$ ,  $\arg z = 0$ .
- b. Dacă  $y = 0$  și  $x$  este diferit de zero,  
atunci  
dacă  $x > 0$ ,  $\arg z = 0$ ;  
altfel  
 $\arg z = \pi = 3.14159265358979$ .
- c. Dacă  $x = 0$  și  $y$  este diferit de zero,  
atunci  
dacă  $y > 0$ ,  $\arg z = \pi/2$ ;  
altfel  
 $\arg z = 3\pi/2$ .
- d. Dacă  $x$  și  $y$  sînt diferiți de zero,  
atunci fie:  
 $a = \arctg(y/x)$

- d1. Dacă  $x > 0$  și  $y > 0$ , atunci  $\arg z = a$ .
- d2. Dacă  $x > 0$  și  $y < 0$ , atunci  $\arg z = 2\pi + a$ .
- d3. Dacă  $x < 0$  și  $y > 0$ , atunci  $\arg z = \pi + a$ .
- d4. Dacă  $x < 0$  și  $y < 0$ , atunci  $\arg z = \pi + a$ .

Se observă că pentru  $x < 0$  și  $y$  diferit de zero:

$\arg z = \pi + a$ ;

altfel, dacă  $x > 0$  și  $y < 0$ , atunci

$\arg z = 2\pi + a$

## FUNCȚIA BX2

```
double darg(COMPLEX *z)
{
    double a;

    if( z->x == 0 && z->y == 0)
        return 0.0;
    if( z->y == 0)
        if( z->x > 0)
            return 0.0;
        else /*y=0 si x<0*/
            return PI;
    if(z->x == 0)
        if(z->y > 0)
            return PI/2;
        else /*x=0 si y<0*/
            return (3*PI)/2;

    /*x!=0 si y!=0*/
    a = atan(z->y/z->x);
    if(z->x < 0 ) /*x<0 si y!=0*/
        return a + PI;
    else /*x>0*/
        if( z->y < 0 ) /*x>0 si y<0*/
            return 2*PI + a;

    /*x>0 si y>0*/
    return a;
}
```

- 10.3 Să se scrie un program care citește numere complexe și le afișează împreună cu modulul și argumentul lor.

### PROGRAMUL BX3

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

typedef struct {
    double x;
    double y;
} COMPLEX;

#include "bx1.cpp"
#include "bx2.cpp"

main() /* citește numere complexe și le afișează împreună cu modulul
      și argumentul corespunzător */
{
    COMPLEX complex;

    while(scanf("%lf %lf",
                &complex.x, &complex.y) == 2 ){
        printf("a+ib= %g + i*(%g)\n", complex.x,
               complex.y);
        printf("modul=%g\targ=%g\n",
               dmodul(&complex), darg(&complex));
    }
}
```

- 10.4 Să se scrie o funcție care afișează un caracter într-un punct de coordonate date.

Ecranul, în mod text, poate fi gestionat folosind funcții care au prototip în fișierul *conio.h*. Astfel, pentru a poziționa cursorul în punctul de coordonate (x,y), se va apela funcția *gotoxy*. Prototipul acestei funcții este:

```
void gotoxy (int x, int y);
```

Pentru a afișa, începând cu punctul poziționat cu ajutorul funcției *gotoxy*, se pot folosi diferite funcții, ca de exemplu, *putch* sau *cprintf* în locul funcției *printf*. Funcția *cprintf* are aceiași parametri ca și funcția *printf*. Prototipul ei se află în fișierul *conio.h* spre deosebire de al funcției *printf* care se află în

*stdio.h.*

O diferență a funcției *cprintf* față de funcția *printf* constă în aceea că '\n' se interpretează diferit. Astfel, apelul:

```
cprintf("\n");
```

trece cursorul pe rîndul următor lăsîndu-l în aceeași coloană. De aceea, apelul:

```
printf("\n");
```

se echivalează cu apelul:

```
cprintf("\n\r");
```

dacă cursorul nu este pe ultimul rînd (25).

Funcția *cprintf* nu realizează defilarea ecranului și de aceea, caracterul '\n' se neglijează dacă cursorul se află pe ultimul rînd.

## FUNCȚIA BX4

```
void afiscar(PUNCT *p, char c)
```

```
/* - afiseaza caracterul c in punctul de coordonate (p->x,p->y);
```

```
   - se presupune ca punctul nu este in afara ecranului. */
```

```
{
    gotoxy( p->x, p->y );
    putch(c);
}
```

10.5 Să se scrie o funcție care verifică dacă un punct are coordonatele în limitele ecranului (25 de linii a 80 coloane).

Funcția returnează valoarea 1 dacă punctul se află în limitele ecranului și 0 în caz contrar.

## FUNCȚIA BX5

```
int limecr(PUNCT *p)
```

```
/* returneaza 1 daca punctul de coordonate (p->x,p->y) se afla in
   limitele ecranului si zero altfel */
```

```
{
    if(p->x <= 0 || p->x > 80)
        return 0;
    if(p->y <= 0 || p->y > 25)
        return 0;
    return 1;
}
```

}

- 10.6 Să se scrie o funcție care trasează un segment de dreaptă orizontal afișând repetat un același caracter dat.

Se presupune că limitele segmentului se află pe ecran.

### **FUNCȚIA BX6**

```
void segoriz(PUNCT *a, PUNCT *b, char c)
/* - traseaza un segment de dreapta orizontal prin afisarea repetata a
   caracterului c;
   - se presupune ca punctul a precede pe b si ca a->y = b->y */
{
    int i;
    PUNCT crt;

    crt.y = a->y;
    for(i=a->x; i <= b->x; i++){
        crt.x = i;
        afiscar(&crt,c);
    }
}
```

- 10.7 Să se scrie o funcție care trasează un segment de dreaptă vertical afișând repetat un același caracter dat.

Se presupune că limitele segmentului se află pe ecran.

### **FUNCȚIA BX7**

```
void segvert(PUNCT *a, PUNCT *b, char c)
/* - traseaza un segment de dreapta vertical prin afisarea repetata a
   caracterului c;
   - se presupune ca punctul a este deasupra punctu'ui b si ca
   a->x = b->x */
{
    int i;
    PUNCT crt;

    crt.x = a->x;
    for(i = a->y; i <= b->y; i++) {
        crt.y = i;
        afiscar(&crt,c);
    }
}
```

```

}
}

```

- 10.8 Să se scrie o funcție care trasează un segment de dreaptă oblic de pantă 45 de grade sexagesimale afișând repetat un același caracter dat.

## FUNCȚIA BX8

```

void segoblic(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreapta oblic de panta 45 grade sexagesimale
   prin afisarea repetata a caracterului c;
   - se presupune ca punctele a si b permit trasarea unui astfel de seg-
   ment de dreapta si ca ele au coordonate ce apartin ecranului. */
{
    int i,j,k,l;
    int pasi,pask;

    i = a->x;
    j = b->x;
    k = a->y;
    l = b->y;
    if( i < j) /* a in stanga lui b */
        pasi = 1;
    else /* a in dreapta lui b */
        pasi = -1;
    if( k < l) /* a deasupra lui b */
        pask = 1;
    else /* b deasupra lui a */
        pask = -1;

    /* trasarea segmentului */
    for( ; (j-i)*pasi >= 0; i +=pasi, k += pask) {
        gotoxy(i,k);
        putchar(c);
    }
}

```

### Observație:

Segmentul se trasează de la punctul *a* spre punctul *b*. De exemplu, dacă punctul *a* este deasupra lui *b* (*pask* = 1) și în dreapta lui *b* (*pasi* = -1), atunci se observă că *i* descrește pînă cînd *i* < *j*. În acest moment produsul  $(j-i)*pasi < 0$  și deci ciclul *for* se termină.

De fiecare dată cînd  $i$  s-a micșorat cu o unitate,  $k$  a crescut cu o unitate. De aceea, poziția caracterului curent față de cel precedent se află în coloana din stînga și cu o linie mai jos.

Punctele  $a$  și  $b$  trebuie să satisfacă relația:

$$(j-i) \cdot \text{pasi} = (l-k) \cdot \text{pask}$$

și de aceea, cînd  $i=j$ , vom avea și  $k=l$ , adică se ajunge în punctul  $b$ .

10.9 Să se scrie un program care trasează un dreptunghi.

Laturile orizontale se trasează folosind caracterul "-" (minus), iar cele verticale folosind caracterul "|" (bara verticală).

## PROGRAMUL BX9

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef struct {
    int x;
    int y;
} PUNCT;

#include "bx4.cpp" /* afiscar */
#include "bx6.cpp" /* segoriz */
#include "bx7.cpp" /* segvert */
#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */

main() /* citește coordonatele coltului stînga sus și a coltului dreapta
        jos ale unui dreptunghi după care trasează dreptunghiul
        respectiv folosind caracterul
            - pentru laturile orizontale și
            | pentru cele verticale. */
{
    PUNCT v1, v2, v;
    char er[] = "s-a tastat EOF\n";

    /* citește coordonatele celor două colturi ale dreptunghiului */
    /* coltul din stînga sus */
    if (pcit_int_lim("x_stanga_sus: ", 1, 79,
        &v1.x) == 0) {
```



```

        printf(er);
        exit(1);
    }
    if(pcit_int_lim("y_stanga_sus: ",1,24,
        &v1.y) == 0 ) {
        printf(er);
        exit(1);
    }

/* coltul din dreapta jos */
    if(pcit_int_lim("x_dreapta_jos: ",v1.x+1,80,
        &v2.x) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("y_dreapta_jos: ",v1.y+1,25,
        &v2.y) == 0 ) {
        printf(er);
        exit(1);
    }

/* sterge ecranul */
    clrscr();

/* trasarea laturilor dreptunghiului */
/* laturile orizontale */
/* latura de sus */
    v.x = v2.x;
    v.y = v1.y;
    segoriz(&v1,&v,'-');

/* latura de jos */
    v.x = v1.x;
    v.y = v2.y;
    segoriz(&v, &v2,'-');

/* laturile verticale */
    v1.y++;
    v2.y--;
    if(v1.y > v2.y )
        exit(0); /* nu sint laturi verticale */

/* latura din stinga */

```

```

v.x = v1.x;
v.y = v2.y;
segvert(&v1, &v, ' | ');

/* latura din dreapta */
v.x = v2.x;
v.y = v1.y;
segvert(&v, &v2, ' | ');
getch();
}

```

10.10 Să se scrie un program care afișează figura de mai jos.



Această figură se compune din segmente de dreaptă orizontale, verticale și oblice de pantă 45 de grade sexagesimale. De aceea, ea se poate afișa folosind funcțiile *segoriz*, *segvert* și *segoblic*.

Virful de sus are coordonatele A(40,1). Segmentul vertical din stînga are extremitățile B(35,6) și C(35,16).

Virful de jos are coordonatele D(40,21). Segmentul vertical din dreapta are extremitățile F(45,6) și E(45,16).

Segmentul orizontal de sus are extremitățile B și F, iar cel de jos extremitățile C și E.

Segmentul orizontal, care este axă de simetrie a figurii, are extremitățile G(35,11) și H(45,11).

În afară de aceste segmente, figura mai conține următoarele segmente oblice:

AB, AF, BE, FC, CD și ED.

## PROGRAMUL B31.C

```
#include <conio.h>

typedef struct {
    int x;
    int y;
} PUNCT;

#include "bx4.cpp" /* afiscar */
#include "bx6.cpp" /* segoriz */
#include "bx7.cpp" /* segvert */
#include "bx8.cpp" /* segoblic */

#define C '*'

main () /* afiseaza o figura formata din segmente orizontale, verticale
        si oblice folosind caracterul *. */
{
    static PUNCT a = {40,1};
    static PUNCT b = {35,6};
    static PUNCT c = {35,16};
    static PUNCT d = {40,21};
    static PUNCT e = {45,16};
    static PUNCT f = {45,6};
    static PUNCT g = {35,11};
    static PUNCT h = {45,11};

    /* sterge ecranul */
    clrscr();

    /* afiseaza segmentele verticale */
    segvert(&b, &c, C);
    segvert(&a, &d, C);
```

```

    segvert(&f,&e,C);

/* afiseaza segmentele orizontale */
    segoriz(&b,&f,C);
    segoriz(&c,&e,C);
    segoriz(&g,&h,C);

/* afiseaza segmentele oblice */
    segoblic(&a,&b,C);
    segoblic(&a,&f,C);
    segoblic(&b,&e,C);
    segoblic(&f,&c,C);
    segoblic(&c,&d,C);
    segoblic(&e,&d,C);
    getch();
}

```

10.11 Să se scrie un program care realizează următoarele:

- citește datele de identificare și notele obținute de o grupă de candidați la admiterea în învățământul superior;
- listează datele citite împreună cu media notelor obținute de fiecare candidat, media grupei la fiecare materie și media mediilor.

La începutul programului se citește un întreg ce reprezintă numărul examenelor susținute, apoi se citesc denumirile materiilor la care s-au susținut examene. În continuare, pentru fiecare candidat, se citesc:

- nume;
- prenume;
- data nașterii;
- adresa;
- nota obținută la fiecare examen.

Dacă un candidat are mai multe prenume, acestea se concatenează prin liniuța de unire, nu se vor separa prin spații. În felul acesta, chiar dacă sînt mai multe prenume, ele se pot citi prin *scanf*, ca și cum ar fi un singur prenume.

Data nașterii se tastează printr-un șir de 6 cifre:

zzllaa

deci, primele două cifre reprezintă ziua, următoarele două luna, iar ultimele

două anul.

Adresa este o succesiune de caractere care nu conține caractere albe. Spațiile din cadrul unei adrese se vor înlocui prin caracterul subliniere. În felul acesta, adresa poate fi citită ca un singur element.

Programul utilizează următoarele funcții definite în exercițiile precedente:

<i>pcit_int_lim</i>	- Pentru a citi date de tip <i>int</i> care se află într-un interval precizat (exercițiul 8.3).
<i>pcit_data_calend</i>	- Pentru a citi și valida o dată calendaristică (exercițiul 8.4).
<i>pcit_int</i>	- Funcție apelată de <i>pcit_int_lim</i> (exercițiul 8.2).
<i>v_calend</i>	- Funcție apelată de <i>pcit_data_calend</i> (exercițiul 6.5).

Programul mai apelează funcția *pcit\_sir* definită în acest exercițiu, care realizează următoarele:

- afișează un text explicativ;
- citește o succesiune de caractere;
- păstrează succesiunea citită în memoria *heap*;
- returnează pointerul spre începutul zonei din memoria *heap* în care se păstrează caracterele respective.

Programul se execută conform pașilor de mai jos:

1. Citește pe *n*, numărul examenelor (se apelează *pcit\_int\_lim*).
2. Citește denumirile celor *n* materii care se examinează (se apelează *pcit\_sir* de *n* ori).
3. *i* = 0 (numărător pentru candidați).
4. Se citesc datele de identificare ale unui candidat și notele acestuia.  
Se păstrează datele citite, apoi se trece la pasul 5.  
Citirile se realizează apelînd funcțiile indicate mai sus.  
Dacă nu mai sînt date de citit, se trece la punctul 7.
5. *i* = *i* + 1.
6. Se reia de la punctul 4.
7. Se fac inițializări pentru calcule de medii pe materii și a mediei generale
8. Se afișează antetul listei.

9. Pentru fiecare candidat se afișează:

- lista cu datele de identificare a candidatului, notele obținute la fiecare materie (nota 0 se consideră neprezentare la examen) și media notelor respective;

după ce s-au listat toate datele pentru toți candidații, se trece la punctul 10.

10. Se afișează mediile pe materii;

11. Se afișează media generală.

În programul de față se presupune că sînt cel mult 5 examene.

Datele relative la un candidat se păstrează în memoria *heap*. De asemenea, tot în memoria *heap* se păstrează denumirile materilor de examinat.

## PROGRAMUL BX11

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bv15.cpp" /* v_calend */
#include "bviii4.cpp" /* pcit_data_calend */

#define MAX 100

int nrzile[] = {0, 31, 28, 31, 30, 31, 30, 31,
               31, 30, 31, 30, 31};
int pcit_sir(char *, char **);

main() /* listeaza situatia notelor unei grupe de candidati obtinute la
       admiterea in invatamintul superior */
{
    typedef struct {
        int zi;
        int luna;
        int an;
    } DC;
```

```

typedef struct {
    char *nume;
    char *prenume;
    DC data_nasterii;
    char *adresa;
    int nrex;
    int nota[5];
} CAND;

CAND *tcand[MAX];
char *denex[5];
int nex;
char er[]="s-a citit EOF\n";
int i,j,k;
char *temp;
float tmed[5];
float medgen;
float med;

/* citeste numarul examenelor */
if(pcit_int_lim("numarul examenelor:1-5: ",1,5,
               &nex) == 0 ){
    printf(er);
    exit(1);
}

/* citeste denumirile materiilor si le pastreaza in memoria heap */
for(i=0; i<nex; i++)
    if(pcit_sir("denumire materie: ",
               &denex[i]) == 0 ) {
        printf(er);
        exit(1);
    }

/* se citesc datele de identificare ale elevilor si notele */
i = 0;
while( i < MAX ) { /* 1 */
    printf("se citesc datele elevului nr: %d\n",
           i+1);

    /* citeste numele candidatului */
    if(pcit_sir("nume: ",&temp) == 0)
        break; /* nu mai sint alti candidati */
}

```

```

/* rezerva zona pentru o data de tip CAND */
    if((tcand[i]=(CAND *)malloc(sizeof(CAND)))
        == 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }

/* tcand[i] are ca valoare adresa de inceput a zonei de memorie din
   memoria heap in care se pastreaza o data structurata de tip CAND */

/* se pastreaza pointerul spre numele candidatului curent citit, nume
   pastrat de functia pcit_sir in memoria heap */
    tcand[i] -> nume = temp;

/* se citește prenumele */
    if(pcit_sir("prenume: ",
        &tcand[i] -> prenume)== 0 ) {
        printf(er);
        exit(1);
    }

/* citește data nasterii */
    if(pcit_data_calend(
        &tcand[i] -> data_nasterii.zi,
        &tcand[i] -> data_nasterii.luna,
        &tcand[i] -> data_nasterii.an) == 0) {
        printf(er);
        exit(1);
    }

/* citește adresa */
    if(pcit_sir("adresa: ",
        &tcand[i] -> adresa)== 0 ) {
        printf(er);
        exit(1);
    }
    tcand[i] -> nrex = nex; /* pastreaza numarul examen-
                           elor */

/* se citesc notele; se tasteaza in ordinea citirii denumirilor materilor de
   examinat */
    for(j=0; j < nex; j++)

```



```

        if(pcit_int_lim(denex[j],0,10,
            &tcand[i] -> nota[j])==0 ){
            printf(er);
            exit(1);
        }

/* s-au citit toate datele pentru un candidat */
    i++;
} /* sfirsit while 1 */

/* initializari pentru calculul mediilor pe materii */
for(j=0; j < nex; j++) {
    tmed[j] = 0.0;
    medgen = 0.0;

/* afiseaza antetul */
    printf("\n\n\tNotele de la examenul de admitere\
        din 7 septembrie 1994\n");
    printf("\n\n\n");
    printf("%20s", " "); /* 20 spatii pentru nume prenume */

/* se scriu denumirile examenelor */
    for(j=0; j<nex;j++) {
        printf("%10s",denex[j]);
        printf("   ");/* cel putin un spatiu dupa fiecare denumire */
    }
    printf("Media\n\n");

/* lista cu datele de identificare, notele si media pentru
    fiecare candidat */
    for(j=0; j<i; j++) {

/* datele de identificare */
        printf("%25s %25s %02d/%02d/%d\n",
            tcand[j] -> nume,
            tcand[j] -> prenume,
            tcand[j] -> data_nasterii.zi,
            tcand[j] -> data_nasterii.luna,
            tcand[j] -> data_nasterii.an);

/* adresa */
        printf("\t%s\n",tcand[j] -> adresa);
    }
}

```

```

/* note */
    printf("%20s", " ");
    for(k=0, med = 0.0; k < nex; k++) {
        printf("%10d ", tcand[j] -> nota[k]);
        med += tcand[j] -> nota[k];
        tmed[k] += tcand[j] -> nota[k];
    }
    printf("%6.2f\n", med/nex);
} /* sfirsit date candidati */

/* afisare media pe materii */
printf("\n\n Media pe materii ");
for(j=0; j<nex; j++) {
    printf("%11.2f", tmed[j]/i);
    medgen += tmed[j] / i;
}

/* afisare media generala */
printf("\n\n Media generala: %10.2f\n",
        medgen / nex);
} /* sfirsit main */

int pcit_sir( char *text, char **sir)
/* - afiseaza text si citeste o succesiune de caractere;
   - pastreaza succesiunea citita in memoria heap si atribuie adresa acestei zone, pointerului sir;
   - returneaza:
       0 - la sfirsit de fisier
       1 - in caz contrar
   - intrerupe executia programului daca nu exista memorie heap
     suficienta pentru a pastra sirul de caractere citit. */
{
    char t[255];
    char *p;

    printf(text);
    if(gets(t) == 0 )
        return 0;

    /* rezerva zona in memoria heap */
    if((p = (char *)malloc(strlen(t) + 1 ))== 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }
}

```

```

    }

/* transfera sirul din t la adresa continuta in p */
strcpy(p,t);
*air = p;
return 1;
}

```

- 10.12 Să se scrie o funcție care schimbă semnul la partea reală și cea imaginară a unui număr complex.

Funcția utilizează tipul COMPLEX introdus prin declarația:

```

typedef struct {
    double x;
    double y;
} COMPLEX;

```

Acest tip se utilizează în exercițiile care urmează în acest capitol.

## **FUNCȚIA BX12**

```

void negcomplex(COMPLEX *a, COMPLEX *b) /* b = -a */
{
    b -> x = - a -> x;
    b -> y = - a -> y;
}

```

- 10.13 Să se scrie o funcție care atribuie partea reală și imaginară a unui număr complex la partea reală și imaginară a unui alt număr complex.

## **FUNCȚIA BX13**

```

void atribcomplex(COMPLEX *a, COMPLEX *b) /* b = a */
{
    b -> x = a -> x;
    b -> y = a -> y;
}

```

- 10.14 Să se scrie o funcție care adună două numere complexe.

## **FUNCȚIA BX14**

```

void adcomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)
/* c = a+b */

```

```

{
    c -> x = a -> x + b -> x;
    c -> y = a -> y + b -> y;
}

```

10.15 Să se scrie o funcție care scade două numere complexe.

### **FUNCȚIA BX15**

```

void sccomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)
/* c = a - b */
{
    c -> x = a -> x - b -> x;
    c -> y = a -> y - b -> y;
}

```

10.16 Să se scrie o funcție care înmulțește două numere complexe.

Dacă  $a$  și  $b$  sînt două numere complexe de forma:

$$a = ax + i \cdot ay$$

și

$$b = bx + i \cdot by$$

atunci produsul lor se definește astfel:

$$a \cdot b = (ax + i \cdot ay) \cdot (bx + i \cdot by) = ax \cdot bx - ay \cdot by + (ax \cdot by + bx \cdot ay) \cdot i$$

### **FUNCȚIA BX16**

```

void mulcomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)
/* c = a * b */
{
    c -> x = a -> x * b -> x - a -> y * b -> y;
    c -> y = a -> x * b -> y + b -> x * a -> y;
}

```

10.17 Să se scrie o funcție care împarte două numere complexe. Ea returnează valoarea zero dacă împărțitorul este nul și unu în caz contrar.

Dacă  $a$  și  $b$  sînt două numere complexe de forma:

$$a = ax + i \cdot ay$$

și

$$b = bx + i \cdot by$$

atunci:

$$\begin{aligned} c = a/b &= (ax + i \cdot ay)/(bx + i \cdot by) = \\ &= [(ax + i \cdot ay)(bx - i \cdot by)]/(bx \cdot bx + by \cdot by) = \\ &= (ax \cdot bx + ay \cdot by)/(bx \cdot bx + by \cdot by) + \\ &\quad i \cdot (ay \cdot bx - ax \cdot by)/(bx \cdot bx + by \cdot by) \end{aligned}$$

## FUNCȚIA BX17

```
int divcomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)
/* c = a/b */
{
    double d;

    d = b->x * b->x + b->y * b->y;
    if( d == 0 )
        return 0;
    c->x = (a->x* b->x + a->y * b->y)/d;
    c->y = (a->y* b->x - a->x* b->y)/d;
    return 1;
}
```

10.18 Să se scrie o funcție care afișează un text și citește un număr de tip *DOUBLE*.

Funcția returnează zero dacă se întâlnește sfârșitul de fișier și unu în caz contrar.

Această funcție este similară cu funcția *pcu\_int* definită în exercițiul 8.2.

## FUNCȚIA BX18

```
int pcit_double(char text[], double *x)
/* - afiseaza caracterele pastrate in tabloul text, citeste un numar de tip
   double si-l pastreaza in zona de memorie spre care pointeaza x;
   - returneaza:
       0 - la intilnirea sfirsitului de fisier;
       1 - altfel. */
{
    char t[255];

    for ( ; ; ) {
        printf(text);
```

```

    if(gets(t) == 0 )
        return 0;
    if(sscanf(t,"%lf",x) == 1)
        return 1;
    printf("nu s-a tastat un numar\n");
}
}

```

- 10.19 Să se scrie o funcție care citește partea reală și partea imaginară a unui număr complex.

Funcția returnează zero la întâlnirea sfârșitului de fișier și unu în caz contrar.

### FUNCȚIA BX19

```

int pcitcomplex(COMPLEX *a)
/* - citește partea reală și cea imaginară a numărului complex a;
   - returnează:
       0 - la întâlnirea sfârșitului de fișier;
       1 - altfel. */
{
    if(pcit_double("partea reală: ", &a -> x) == 0 )
        return 0;
    if(pcit_double("partea imaginară: ", &a -> y) == 0 )
        return 0;
    return 1;
}

```

- 10.20 Să se scrie un program care citește trei numere complexe  $a, b, c$  care sînt coeficienții ecuației:

$$a \cdot x^2 + b \cdot x + c = 0$$

rezolvă și afișează rădăcinile ecuației respective.

### PROGRAMUL BX20

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.14159265358979

```

```

typedef struct {
    double x;
    double y;
} COMPLEX;

#include "bx12.cpp" /* negcomplex */
#include "bx14.cpp" /* adcomplex */
#include "bx15.cpp" /* scomplex */
#include "bx16.cpp" /* mulcomplex */
#include "bx17.cpp" /* divcomplex */
#include "bx1.cpp" /* dmodul */
#include "bx2.cpp" /* darg */
#include "bx18.cpp" /* pcit_double */
#include "bx19.cpp" /* pcitcomplex */

main() /* - citește pe a,b,c - numere complexe;
        - rezolvă și afișează rădăcinile ecuației:
             $a \cdot x^2 + b \cdot x + c = 0.$  */
{
    COMPLEX a,b,c,x1,x2,bp,temp,aa,temp1;
    char er[] = "s-a tastat EOF\n";
    double r,arg;
    COMPLEX patru = {4.0,0.0};

/* se citesc coeficientii*/

    if(pcitcomplex(&a) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcitcomplex(&b) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcitcomplex(&c) == 0 ) {
        printf(er);
        exit(1);
    }
    if( a.x == 0 && a.y == 0 && b.x == 0 &&
        b.y == 0 && c.x == 0 && c.y == 0 ) {

/* a = b = c = 0 */
        printf("ecuație nedeterminată\n");
    }
}

```

```

        exit(0);
    }
    if( a.x == 0 && a.y == 0 && b.x == 0 &&
        b.y == 0 ) {

/* a=b=0 sic != 0 */
        printf("ecuatia nu are solutie\n");
        exit(1);
    }
    if(a.x == 0 && a.y == 0 ) {

/* ecuatia de gradul 1 */
        negcomplex(&c,&x1);
        divcomplex(&x1,&b,&x2);
        printf("ecuatia de gradul 1\n");
        printf("x=%g + i*(%g)\n", x2.x, x2.y);
        exit(0);
    } /* temp = b*b - 4*a*c */

    mulcomplex(&b,&b,&bp);
    mulcomplex(&patru,&a,&temp);
    mulcomplex(&temp,&c,&templ);
    sccomplex(&bp,&templ,&temp);

/*temp = x + i*y
   r = modul(temp)
   arg = argument(temp) */

    r = dmodul(&temp);
    arg = darg(&temp);

/* sqrt(temp); temp = r*(cos(arg) + i*sin(arg));
   sqrt(temp)=sqrt(r)*(cos(arg/2)+i*sin(arg/2)) */

    r = sqrt(r);
    arg = arg/2;

/* sqrt(b*b - 4*a*c) = r*(cos(arg)+i*sin(arg)) */

    temp.x = r*cos(arg);
    temp.y = r*sin(arg);

```



```

/* aa = 2*a = a+a */

adcomplex(&a, &a, &aa);

/* x1=(-b+sqrt(b*b-4*a*c))/(2*a) */
/* bp=-b */

negcomplex(&b, &bp);
adcomplex(&bp, &temp, &templ);
divcomplex(&templ, &aa, &x1);

/* x2=(-b-sqrt(b*b-4*a*c))/(2*a) */

sccomplex(&bp, &temp, &templ);
divcomplex(&templ, &aa, &x2);
printf("x1=%g+i(%g)\n", x1.x, x1.y);
printf("x2=%g+i(%g)\n", x2.x, x2.y);
}

```

10.21 Să se scrie o funcție care adună două matrice pătrate de ordinul trei. Elementele matricelor sînt numere de tip *double*.

Funcția de față și cele care urmează în acest capitol vor folosi tipul MATP3 declarat astfel:

```

typedef struct {
    double x11,x12,x13,x21,x22,x23,x31,x32,x33;
} MATP3;

```

## FUNCȚIA BX21

```

void admatp(MATP3 *a, MATP3 *b, MATP3 *c) /* c = a + b */
{
    c->x11 = a->x11 + b->x11;
    c->x12 = a->x12 + b->x12;
    c->x13 = a->x13 + b->x13;
    c->x21 = a->x21 + b->x21;
    c->x22 = a->x22 + b->x22;
    c->x23 = a->x23 + b->x23;
    c->x31 = a->x31 + b->x31;
    c->x32 = a->x32 + b->x32;
    c->x33 = a->x33 + b->x33;
}

```

10.22 Să se scrie o funcție care înmulțește la stînga o matrice pătratică de ordinul trei cu un vector de trei elemente.

Dacă matricea are elementele:

x11 x12 x13

x21 x22 x23

x31 x32 x33

iar vectorul este  $b$  de componente:

$b[0], b[1], b[2]$

atunci produsul dintre matrice și vectorul  $b$  este un vector  $c$  de trei elemente:

$c[i-1] = x_{i1} * b[0] + x_{i2} * b[1] + x_{i3} * b[2]$  pentru  $i = 1, 2, 3$ .

### **FUNCȚIA BX22**

```
void mulmatvect3(MATP3 *a, double b[], double c[])
/* c = a*b */
{
    c[0] = a->x11 * b[0] + a->x12 * b[1] +
           a->x13 * b[2];
    c[1] = a->x21 * b[0] + a->x22 * b[1] +
           a->x23 * b[2];
    c[2] = a->x31 * b[0] + a->x32 * b[1] +
           a->x33 * b[2];
}
```

10.23 Să se scrie o funcție care înmulțește două matrice pătratice de ordinul trei.

### **FUNCȚIA BX23**

```
void mulmatp3(MATP3 *a, MATP3 *b, MATP3 *c )
/* c = a * b */
{
    c -> x11 = a -> x11 * b -> x11 + a -> x12 *
              b -> x21 + a -> x13 * b -> x31;
    c -> x12 = a -> x11 * b -> x12 + a -> x12 *
              b -> x22 + a -> x13 * b -> x32;
    c -> x13 = a -> x11 * b -> x13 + a -> x12 *
              b -> x23 + a -> x13 * b -> x33;
    c -> x21 = a -> x21 * b -> x11 + a -> x22 *
```

```

        b -> x21 + a -> x23 * b -> x31;
c -> x22 = a -> x21 * b -> x12 + a -> x22 *
    b -> x22 + a -> x23 * b -> x32;
c -> x23 = a -> x21 * b -> x13 + a -> x22 *
    b -> x23 + a -> x23 * b -> x33;
c -> x31 = a -> x31 * b -> x11 + a -> x32 *
    b -> x21 + a -> x33 * b -> x31;
c -> x32 = a -> x31 * b -> x12 + a -> x32 *
    b -> x22 + a -> x33 * b -> x32;
c -> x33 = a -> x31 * b -> x13 + a -> x32 *
    b -> x23 + a -> x33 * b -> x33;
}

```

- 10.24 Să se scrie o funcție care negativează elementele unei matrice pătratice de ordinul 3.

#### FUNCȚIA BX24

```

void negmatp3(MATP3 *a, MATP3 *b) /* b = -a */
{
    b -> x11 = - a -> x11;
    b -> x12 = - a -> x12;
    b -> x13 = - a -> x13;
    b -> x21 = - a -> x21;
    b -> x22 = - a -> x22;
    b -> x23 = - a -> x23;
    b -> x31 = - a -> x31;
    b -> x32 = - a -> x32;
    b -> x33 = - a -> x33;
}

```

- 10.25 Să se scrie o funcție care calculează și returnează valoarea determinantului unei matrice pătratice de ordinul 3.

Calculul determinantului se face folosind regula lui *Saarus*.

#### FUNCȚIA BX25

```

double detmatp3(MATP3 *a)
/* calculeaza si returneaza valoarea determinantului matricei a */
{
    return a -> x11 * a -> x22 * a -> x33 +
        a -> x12 * a -> x23 * a -> x31 +
        a -> x21 * a -> x32 * a -> x13 -

```

```

a -> x31 * a -> x22 * a -> x13 -
a -> x11 * a -> x32 * a -> x23 -
a -> x21 * a -> x12 * a -> x33;

```

```

}

```

10.26 Să se scrie o funcție care calculează inversa unei matrice pătratice de ordinul 3. Funcția returnează 0 dacă matricea are determinantul nul și unu în caz contrar.

Fie  $a$  matricea de ordinul 3:

```

x11 x12 x13
x21 x22 x23
x31 x32 x33

```

Inversa matricei  $a$  este matricea:

```

X11 X21 X31
X12 X22 X32
X13 X23 X33

```

unde prin  $X_{ij}$  am notat complementul algebric al elementului  $a_{ij}$  al matricei date. Dacă notăm cu  $d$  determinantul matricei  $a$ , atunci  $X_{ij}$  se calculează astfel:

$X_{ij} = A_{ij} / d$     dacă  $i+j$  este par;

și

$X_{ij} = -A_{ij} / d$     dacă  $i+j$  este impar,

unde prin  $A_{ij}$  s-a notat determinantul matricei obținut din matricea inițială suprimind linia  $i$  - a și coloana  $a_j$  - a.

## FUNCȚIA BX26

```

int invmatp3(MATP3 *a, MATP3 *b )
/* - în b se obține inversa lui a;
   - returneaza:
       0 - daca determinantul matricei este 0;
       1 - în caz contrar. */
{
    double d;

    if(( d = detmatp3( a )) == 0 )
        return 0;
    b->x11=(a->x22 * a->x33 - a->x23 * a->x32)/d;
    b->x12=(a->x32 * a->x13 - a->x12 * a->x33)/d;

```

```

b->x13=(a->x12 * a->x23 - a->x13 * a->x22)/d;
b->x21=(a->x31 * a->x23 - a->x21 * a->x33)/d;
b->x22=(a->x11 * a->x33 - a->x31 * a->x13)/d;
b->x23=(a->x21 * a->x13 - a->x11 * a->x23)/d;
b->x31=(a->x21 * a->x32 - a->x31 * a->x22)/d;
b->x32=(a->x12 * a->x31 - a->x11 * a->x32)/d;
b->x33=(a->x11 * a->x22 - a->x12 * a->x21)/d;
return 1;
}

```

10.27 Să se scrie o funcție care citește elementele unei matrice pătratice de ordinul 3 de tip *double*.

Funcția returnează zero la întâlnirea sfârșitului de fișier și unu în caz contrar.

### FUNCȚIA BX27

```

int citelem(double *x, int i, int j); /* prototip */
int citmatp3(MATP3 *a)
/* - citește elementele unei matrice patratice de ordinul 3;
   - returnează:
       0 - la întâlnirea sfârșitului de fișier;
       1 - în caz contrar. */
{
    double l;

    if(citelem(&l,1,1) == 0 )
        return 0;
    else
        a->x11 = l;
    if(citelem(&l,1,2) == 0 )
        return 0;
    else
        a->x12 = l;
    if(citelem(&l,1,3) == 0 )
        return 0;
    else
        a->x13 = l;
    if(citelem(&l,2,1) == 0)
        return 0;
    else
        a->x21 = l;
    if(citelem(&l,2,2) == 0 )

```

```

        return 0;
    else
        a->x22 = 1;
    if(citelem(&l,2,3) == 0 )
        return 0;
    else
        a->x23 = 1;
    if(citelem(&l,3,1) == 0 )
        return 0;
    else
        a->x31 = 1;
    if(citelem(&l,3,2) == 0 )
        return 0;
    else
        a->x32 = 1;
    if(citelem(&l,3,3) == 0 )
        return 0;
    else
        a->x33 = 1;
    return 1;
} /* sfirsit functia citmatp3 */

int citelem( double *x, int i,int j)
/* - citeste un numar de tip double care reprezinta elementul unei
matrice din linia i si coloana j;
- returneaza:
    0 - la intalnirea sfirsitului de fisier;
    1 - altfel. */
{
    char t[255];

    for( ; ; ) {
        printf("elementul din linia %d si\
coloana %d: ",i,j );
        if(gets(t) == 0 )
            return 0;
        if(sscanf(t,"%lf",x) == 1 )
            return 1;
        printf("nu s-a tastat un numar\n");
    }
}

```

10.28 Să se scrie un program care rezolvă un sistem de trei ecuații cu trei

necunoscute de forma:

$$(1) A \cdot x = b$$

Programul de față rezolvă sistemul înmulțind relația (1) la stînga cu inversa lui  $A$ . Deci:

$$x = \text{inv}(A) \cdot b$$

Pașii programului sînt:

1. Citește elementele matricei  $A$   
(funcția *citmatp3*, exercițiul 10.27).
2. Citește termenul liber  $b$   
(funcția *pndcit*, exercițiul 8.8).
3. Inversează matricea  $A$   
(funcția *invmatp3*, exercițiul 10.26).
4. Calculează produsul la stînga dintre inversa matricei  $A$  și termenul liber  $b$   
(funcția *mulmatvect3*, exercițiul 10.22).
5. Afișează soluția obținută  $x0$ .
6. Afișează reziduurile, adică diferența:

$$r = A \cdot x0 - b.$$

În acest scop se realizează:

- produsul  $A \cdot x0$   
(funcția *mulmatvect3*, exercițiul 10.22);
- diferența:  
 $A \cdot x0 - b.$

## PROGRAMUL BX28

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    double x11,x12,x13,x21,x22,x23,x31,x32,x33;
} MATP3;

#include "bviii8.cpp" /* pndcit */
```

```

#include "bx22.cpp" /* mulmatvect3 */
#include "bx25.cpp" /* detmatp3 */
#include "bx26.cpp" /* invmatp3 */
#include "bx27.cpp" /* citmatp3 */

main() /* - rezolva un sistem de 3 ecuatii cu 3 necunoscute inmatrind
        la stanga cu inversa matricei sistemului,
        - afiseaza solutia determinata si reziduurile */
{
    MATP3 a, inva;
    int i;
    double b[3], x0[3], r[3];
    char er[] = "s-a tastat EOF\n";

    /* citeste matricea sistemului */
    if(citmatp3(&a) == 0 ) {
        printf(er);
        exit(1);
    }

    /* citeste termenul liber */
    if(pndcit(3, b) != 3 ) {
        printf(er);
        exit(1);
    }

    /* inverseaza matricea sistemului */
    if(invmatp3(&a, &inva) == 0 ) {
        printf("determinantul sistemului este nul\n");
        exit(1);
    }

    /* produsul la stanga dintre inversa matricei a si termenul liber b */
    mulmatvect3(&inva, b, x0);

    /* afisarea solutiei */
    for( i=0; i < 3; i++)
        printf("x%d= %g\n", i+1, x0[i]);

    /* calculeaza reziduurile */
    /* calculeaza produsul a*x0 */
    mulmatvect3(&a, x0, r);
}

```



```

/* se afiseaza reziduurile */
for( i = 0; i < 3; i++)
    printf("reziduu linia %d= %g\n",
           i+1, r[i] - b[i] );
}

```

#### Observații:

1. Toate funcțiile din exercițiile 10.21 - 10.28, pot fi reserise folosind tablouri de tip *double* în locul structurilor de tip MATP3. Propunem cititorului să facă acest lucru, funcțiile simplificându-se pe baza utilizării instrucțiunilor ciclice.
2. Funcția *culetem* poate fi scrisă mai simplu dacă instrucțiunea

```

if(citelem(&e,i,j) == 0 )
    return 0;
else
    a -> xij = e; pentru i,j = 1,2,3

```

se înlocuiește cu:

```

if(citelem(&a -> xij,i,j) == 0)
    return 0;

```

- 10.29 Să se scrie o funcție care calculează și returnează cel mai mare divizor comun a două numere *m* și *n* de tip *long*

Funcția utilizează algoritmul lui Euclid (vezi exercițiul 4.24).

#### FUNCȚIA BX29

```

long cmmdc(long m, long n)/* calculeaza si returneaza (m,n) *.
{
    long r;

    do {
        r=m%n;
        if(r) {
            m = n;
            n = r;
        }
    } while(r);
    return n;
}

```

### Observație:

Funcția presupune că  $n$  este diferit de zero.

10.30 Să se scrie o funcție care calculează și returnează cel mai mic multiplu comun a două numere  $m$  și  $n$  de tip *long*.

Dacă notăm cu:

$[m,n]$

cel mai mic multiplu comun al numerelor  $m$  și  $n$ , atunci:

$$[m,n] = m * n / (m,n)$$

unde prin:

$(m,n)$  - Am notat cel mai mare divizor comun al aceluiași numere.

### FUNCȚIA BX30

```
long cmmmc(long m, long n) /* calculeaza si returneaza [m,n] */
{
    return m*n/cmmdc(m,n);
}
```

### Observație:

Funcția presupune că ambele numere sunt diferite de zero.

10.31 În acest exercițiu și în următoarele din acest capitol, vom utiliza tipul **RATIONAL** definit astfel:

```
typedef struct {
    long numerator;
    long numitor;
} RATIONAL;
```

O dată de tip **RATIONAL** o vom numi *fracție*. Valoarea unei astfel de date este egală cu raportul:

numerator / numitor, dacă *numitor* este diferit de zero

și

nu este definită, dacă *numitor* are valoarea zero

O fracție este *irreducibilă* dacă numărătorul și numitorul ei sunt numere

prime între ele. Pentru ca o fracție să fie ireductibilă, este suficient să simplificăm fracția respectivă cu cel mai mare divizor comun al numărătorului și numitorului ei. Funcția de față transformă o dată de tip RATIONAL într-o fracție ireductibilă.

## FUNCȚIA BX31

```
void simplifica(RATIONAL *a)
/* simplifica pe a cu (numarator,numitor) */
{
    long d;

    if(a -> numarator == 0 || a -> numitor == 0)
        return;
    d = cmmdc(a->numarator,a->numitor);
    a -> numarator /= d;
    a -> numitor /= d;
}
```

10.32 Să se scrie o funcție care adună două date de tip RATIONAL.

Fie  $a$  și  $b$  cele două date.

Adunarea se execută conform următorilor pași:

1. Dacă numărătorul lui  $a$  este zero, rezultatul este  $b$  și se revine din funcție.
2. Dacă numărătorul lui  $b$  este zero, rezultatul este  $a$  și se revine din funcție.
3. Se calculează cel mai mic multiplu comun al numitorilor lui  $a$  și  $b$ , fie acesta  $m$ .
4. Se calculează:  

$$a1 = (\text{numărătorul lui } a) * (m / (\text{numitorul lui } a)).$$
5. Se calculează  $b1$  ca și  $a1$ , folosind numărătorul și numitorul lui  $b$ .
6. Numărătorul rezultatului este suma:  

$$a1 + b1.$$
7. Numitorul rezultatului este  $m$ .
8. Se simplifică rezultatul dacă este posibil.

## **FUNCȚIA BX32**

```
void adfr(RATIONAL *a, RATIONAL *b, RATIONAL *c)
/* c = a + b */
{
    long m;

    if(a -> numitor == 0 || b -> numitor == 0)
        return ;
    if(a->numarator == 0 ){
        c->numarator = b->numarator;
        c->numitor = b->numitor;
    }
    else
        if(b->numarator == 0) {
            c->numarator = a->numarator;
            c->numitor = a->numitor;
        }
        else {
            m = cmucmc(a->numitor, b->numitor);
            c->numarator = a->numarator * (m/a->numitor)
                + b->numarator * (m/b->numitor);
            c->numitor = m;
        }
    simplifica(c);
}
```

10.33 Să se scrie o funcție care scade două date de tip RATIONAL.

## **FUNCȚIA BX33**

```
void subfr(RATIONAL *a, RATIONAL *b, RATIONAL *c)
/* c = a - b */
{
    RATIONAL temp;
    /* temp = -b */

    temp.numarator = -b->numarator;
    temp.numitor = b->numitor;
    adfr(a, &temp, c);
}
```

10.34 Să se scrie o funcție care înmulțește două date de tip RATIONAL.

### **FUNCȚIA BX34**

```
void mulfr(RATIONAL *a,RATIONAL *b,RATIONAL *c)
/* c = a*b */
{
    c->numerator = a->numerator * b->numerator;
    c->numitor = a->numitor * b->numitor;
    simplifica(c);
}
```

10.35 Să se scrie o funcție care împarte două date de tip RATIONAL.

### **FUNCȚIA BX35**

```
void divfr(RATIONAL *a,RATIONAL *b,RATIONAL *c)
/* c = a/b */
{
    c->numerator = a->numerator * b->numitor;
    c->numitor = a->numitor * b->numerator;
    simplifica(c);
}
```

10.36 Să se scrie o funcție care citește numărătorul și numitorul unei date de tip RATIONAL.

### **FUNCȚIA BX36**

```
int citfr(RATIONAL *a)
/* - citește numărătorul și numitorul unei fracții;
   - returnează:
       0 - la întâlnirea sfîrșitului de fișier;
       1 - altfel. */
{
    char t[255];

    for ( ; ; ) {
        printf("numărătorul fracției: ");
        if(gets(t) == 0 )
            return 0;
        if(sscanf(t,"%ld",&a->numerator) == 1)
            break;
        printf("nu s-a tastat un întreg\n");
    }
    for( ; ; ) {
```

```

printf("numitorul fractiei: ");
if(gets(t) == 0)
    return 0;
if(sscanf(t,"%ld",&a->numitor) == 1 &&
    a->numitor != 0 )
    break;
printf("nu s-a tastat un intreg nenul\n");
}
return 1;
}

```

- 10.37 Să se scrie un program care citește componentele a trei date de tip RATIONAL, f1, f2 și f3, calculează și afișează sub formă de fracție valoarea expresiei:

$$(f1 - f2) * f3$$

### PROGRAMUL BX37

```

#include <stdio.h>
#include <stdlib.h>

```

```

typedef struct {
    long numarator;
    long numitor;
} RATIONAL ;

```

```

#include "bx29.cpp" /* cmmdc */
#include "bx30.cpp" /* cmmmc */
#include "bx31.cpp" /* simplifica */
#include "bx32.cpp" /* adfr */
#include "bx33.cpp" /* subfr */
#include "bx34.cpp" /* mulfr */
#include "bx36.cpp" /* citfr */

```

```

main() /* citeste datele f1,f2 si f3 de tip RATIONAL, calculeaza si
        afiseaza valoarea expresiei:

```

$$(f1-f2)*f3$$

```

        sub forma de fractie. */

```

```

{
    RATIONAL f[3],temp;
    char er[] = "s-a tastat EOF\n";
    char *text[]={"prima fractie\n",
                  "fractia a doua\n",

```

```

    "fractia a treia\n"
};
int i;

/* citeste fractiile: f[0]=f1, f[1]=f2, f[2]=f3 */
for(i=0; i<=2 ; i++) {
    printf(text[i]);
    if(citfr(&f[i]) == 0 ) {
        printf(er);
        exit(1);
    }
}

/* temp = f1 - f2 */
subfr(&f[0], &f[1], &temp);

/* temp = temp*f3 */
mulfr(&temp, &f[2], &temp);

/* scrie rezultatul */
printf("(f1-f2)*f3:numarator= %ld\n",
        temp.numarator);
printf("(f1-f2)*f3:numitor= %ld\n", temp.numitor);
}

```

## 10.4. Reunione

Limbajul C oferă utilizatorului posibilitatea de a păstra într-o zonă de memorie date de tipuri diferite. Până în prezent am văzut că unei date *i* se alocă o zonă de memorie potrivit tipului datei respective și în zona alocată ei se pot păstra numai date de acel tip. De exemplu, dacă avem în vedere declarația:

```
long x;
```

atunci pentru *x* se alocă 32 de biți și în zona respectivă se păstrează întregi reprezentați prin complement față de 2.

Se pot ivi situații în care în momente diferite ale execuției, am dori ca în aceeași zonă de memorie să putem păstra date de tipuri diferite. De exemplu, dacă după un timp nu mai este nevoie de variabila *x*, zona alocată lui *x* ar putea fi utilizată în alte scopuri, pentru a păstra o dată de un alt tip: *char*, *int* sau chiar *float*. Astfel de *reutilizări* ale zonelor de memorie pot conduce la economisirea memoriei.

Aceasta este posibil "grupînd" împreună datele care dorim să fie alocate în aceeași zonă de memorie. Pentru a realiza o astfel de grupare se folosește o construcție similară cu cea pentru *structuri*, diferența constînd în aceea că se schimbă cuvîntul *struct* cu *union*. În rest, toate formatele întîlnite în cazul structurilor rămîn valabile. Tipul introdus prin *union* este un tip *definit* de utilizator ca și cel definit prin *struct*. O astfel de dată grupată o vom numi *reuniune*.

#### Exemple:

1.

```
union a {  
    int x;  
    long y;  
    double r;  
    char c;  
} p;
```

*p* este o reuniune de tipul *a*.

Componentele *x*, *y*, *r* și *c* ale lui *p* le referim ca și în cazul structurilor prin:

*p.x*, *p.y*, *p.r* și *p.c*

Ele sînt alocate în aceeași zonă de memorie și de aceea, la un moment dat al execuției, numai una din aceste componente este definită (alocată).

Pentru *p* se alocă o zonă de memorie suficientă pentru a păstra data care necesită numărul maxim de octeți, deci se vor alocă 8 octeți necesari pentru a putea păstra componenta *r* de tip *double*.

Să observăm că dacă înlocuim cuvîntul *union* prin *struct*, atunci pentru *p* se alocă:

- 2 octeți pentru *x*;
- 4 octeți pentru *y*;
- 8 octeți pentru *r*;
- 1 octet pentru *c*,

deci în total 15 octeți. Aceasta deoarece, în cazul unei structuri, componentele ei sînt definite simultan și deci trebuie să fie alocate în zone diferite de memorie.

2.

```
typedef union {  
    char nume[70];
```



```

    int nrmat;
    long cod;
} ZC;

```

```

ZC sir;

```

*sir* este o reuniune de tip *ZC* pentru care s-au alocat 70 de octeți. În această zonă se pot păstra șiruri de caractere la care ne putem referi prin:

```

sir.nume;
sir.nume[0];
sir.nume[1];
etc.

```

sau întregi de tip *int* sau *long*. La aceștia ne referim prin:

```

sir.nrmat

```

și

```

sir.cod.

```

Fie:

```

ZC *p;

```

atunci putem realiza o atribuire de forma:

```

p = &sir;

```

Prin intermediul lui *p* ne putem referi la componentele reuniunii *sir* astfel:

```

p -> nume;
p -> nume[0];
p -> nume[1]
etc.
p -> nrmat

```

și

```

p -> cod.

```

În legătură cu reuniunile, pot apărea probleme la utilizarea lor, deoarece programatorul trebuie să cunoască, în fiecare moment al execuției, ce componentă a reuniunii este prezentă în zona alocată ei.

Fie reuniunea:

```

union {
    int i;
    float f;
}

```

```
double d;
} zc;
```

Dacă la un moment dat, în zona alocată datei *zc* se păstrează un întreg de tip *int* (de exemplu: se face atribuirea *zc.i = 10*) și se fac referiri la componenta *d*:

```
if( zc.d > 0 )
```

atunci rezultatul comparării va fi eronat. Această eroare nu poate fi semnalată de compilator și nici la execuție și de aceea, astfel de utilizări pot fi evitate numai de programator.

Pentru a înlătura erorile de acest fel, se recomandă utilizarea unui indicator care să definească tipul datei păstrate în fiecare moment în zona alocată reuniunii respective. De exemplu, în cazul reuniunii *zc* de mai sus este necesar un indicator care să aibă trei valori corespunzătoare celor trei tipuri ale componentelor lui *zc* (*int*, *float* și *double*). Aceste valori le numim sugestiv prin constante simbolice:

```
#define INTREG 1
#define FSIMPLU 2
#define FDUBLU 3
```

Adăugăm la reuniunea de mai sus un indicator care are una din aceste valori, în funcție de componenta prezentă în zona reuniunii. Se obține tipul utilizator de mai jos:

```
struct tzc {
    int tipert; /* indicator care defineste tipul curent */
    union {
        int i;
        float f;
        double d;
    } zc;
} ;
```

Declarăm structura de tip *tzc*:

```
struct tzc szc;
```

Structura are două componente:

- tipert*                    - Este de tip *int*.
- zc*                        - Este o reuniune de componente *i*, *f* și *d*.

Deci, tot timpul este alocată data *tipert* și numai una din componentele *i*, *f*, *d*.

Pentru a păstra o dată de tip *int*, de exemplu valoarea 123, vom folosi atribuirea:

```
szc.zc.i = 123;
```

Alături de această atribuire vom mai utiliza încă una și anume:

```
szc.tipcrt = INTREG;
```

În felul acesta, componenta *tipcrt* ne permite să stabilim faptul că reuniunea *zc* conține o dată de tip *int*.

În mod analog, dacă dorim să păstrăm, în zona respectivă, o dată flotantă de tip *double*, de exemplu valoarea lui *pi*, vom folosi atribuirile:

```
szc.zc.d = 3.14159265;
```

```
szc.tipcrt = FDUBLU;
```

La utilizarea datelor păstrate în acest fel se poate folosi o construcție *if* de forma:

```
if(szc.tipcrt == INTREG)
    se folosește szc.zc.i
else if(szc.tipcrt == FSIMPLU)
    se folosește szc.zc.f
else if(szc.tipcrt == FDUBLU)
    se folosește szc.zc.d
else
    eroare
```

Aceeași utilizare se obține folosind o instrucțiune *switch*:

```
switch (szc.tipcrt) {
    case INTREG:
        se folosește szc.zc.i
        break;
    case FSIMPLU:
        se folosește szc.zc.f
        break;
    case FDUBLU:
        se folosește szc.zc.d
        break;
    default:
        eroare
}
```

În încheierea acestui paragraf amintim că reuniunile *nu pot* fi inițializate, spre deosebire de structuri.

### Exerciții:

10.38 Fie tipul FIG declarat ca mai jos:

```
typedef struct {
    int tip; /* tipul figurii */
    union {
        double raza; /* cerc */
        double lp; /* pătrat */
        double ld[2]; /* dreptunghi */
        double lt[3]; /* triunghi */
    } fig;
} FIG;
```

O dată de tip FIG conține elementele unei figuri necesare pentru a calcula aria figurii respective. Figurile avute în vedere sînt:

- cerc;
- pătrat;
- dreptunghi

și

- triunghi.

În cazul primelor două figuri, data de tip FIG conține o valoare de tip *double* care, în cazul cercului reprezintă lungimea razei acestuia, iar în cazul pătratului, lungimea laturii pătratului. În cazul dreptunghiului, data conține două elemente de tip *double* lungimea și lățimea. În sfîrșit, în cazul triunghiului, data conține trei valori de tip *double* care reprezintă lungimile celor trei laturi ale triunghiului.

Componenta *tip* definește elementele(figura) prezente într-o dată de tip FIG și are valorile:

- |    |                      |
|----|----------------------|
| 0  | - Pentru cerc.       |
| 1  | - Pentru pătrat.     |
| 2  | - Pentru dreptunghi. |
| 3  | - Pentru triunghi.   |
| -1 | - Pentru eroare.     |

În locul acestor valori, considerăm constantele simbolice:

```
#define EROARE -1
#define CERC 0
#define PATRAT 1
#define DREPTUNGHI 2
```

```
#define TRIUNGHI 3
```

Funcția de mai jos are ca parametru o dată de tip FIG, calculează și returnează aria figurii ale cărei elemente sînt conținute în zona definită de parametru.

Funcția returnează 0 în cazul în care datele sînt eronate.

La calculul ariei unui triunghi se folosește formula lui Heron.

## FUNCȚIA BX38

```
#define PI 3.14159265358979
```

```
double aria(FIG *p)
```

```
/* calculeaza si returneaza aria figurii definite de elementele prezente in  
   zona spre care pointeaza p; la eroare returneaza 0 */
```

```
{  
    double sp,a,b,c;  
  
    switch(p->tip) {  
        case CERC:  
            return PI*p->fig.raza*p->fig.raza;  
        case PATRAT:  
            return p->fig.lp*p->fig.lp;  
        case DREPTUNGHI:  
            return p->fig.ld[0]*p->fig.ld[1];  
        case TRIUNGHI:  
            sp=(p->fig.lt[0] + p->fig.lt[1] +  
                p->fig.lt[2])/2;  
            if((a=sp - p->fig.lt[0]) > 0 &&  
                (b=sp - p->fig.lt[1]) > 0 &&  
                (c=sp - p->fig.lt[2]) > 0 )  
                return sqrt(sp*a*b*c);  
            else { /* cele 3 valori nu reprezinta lungimile laturilor  
                    unui triunghi */  
                printf("a= %g\tb= %g\tc= %g\tnu\  
                        formeaza un triunghi\n",  
                        p->fig.lt[0], p->fig.lt[1],  
                        p->fig.lt[2] );  
                return 0;  
            }  
        default:/* eroare */  
            return 0;  
    }  
}
```

```

    }
}

```

10.39 Să se scrie o funcție care are ca parametru un pointer spre o dată de tip FIG și care citește și păstrează elementele figurii definite de componenta *tip* a datei de tip FIG.

### FUNCȚIA BX39

```

int citfig(FIG *p)
/* - citește elementele figurii definite de p->tip;
   - returnează:
       0 - la întâlnirea sfîrșitului de fișier sau la eroare;
       1 - altfel. */
{
    char t[255];

    switch(p->tip) {
        case CERC:
            for( ; ; ) {
                printf("raza= ");
                if(gets(t) == 0 )
                    return 0;
                if(sscanf(t,"%lf",&p->fig.raza)== 1 &&
                   p->fig.raza>0)
                    return 1;
                printf("nu s-a tastat un numar\
                        pozitiv\n");
            }
        case PATRAT:
            for( ; ; ) {
                printf("latura patratului= ");
                if(gets(t) == 0 )
                    return 0;
                if(sscanf(t,"%lf",&p->fig.lp) == 1 &&
                   p->fig.lp > 0)
                    return 1;
                printf("nu s-a tastat un numar\
                        pozitiv\n");
            }
        case DREPTUNGHI:
            for( ; ; ) {
                printf("lungimea si latimea pe aceeasi\

```

```

        linie: ");
        if(gets(t) == 0 )
            return 0;
        if(sscanf(t,"%lf %lf",&p->fig.ld[0],
            &p->fig.ld[1])==2 &&
            p->fig.ld[0] > 0 &&
            p->fig.ld[1] > 0 )
            return 1;
        printf("nu s-au tastat 2 numere\
            pozitive\n");
    }
    case TRIUNGHI:
        for( ; ; ) {
            printf("laturile triunghiului pe\
                aceeaasi linie: ");
            if(gets(t) == 0 )
                return 0;
            if(sscanf(t,"%lf %lf %lf",&p->fig.lt[0],
                &p->fig.lt[1],
                &p->fig.lt[2])==3 &&
                p->fig.lt[0]>0 &&
                p->fig.lt[1]>0 &&
                p->fig.lt[2]> 0 )
                return 1;
            printf("nu s-au tastat 3 numere\
                pozitive\n");
        }
    default:
        return 0;
    }
}

```

10.40 Să se scrie un program care citește elementele unei figuri necesare pentru calculul ariei sale. Elementele sînt precedate de o literă mare, care definește figura. Această literă se tastează la început, singură pe o linie.

Programul afișează aria figurii.

Correspondența dintre litere și figuri este următoarea:

- |          |                      |
|----------|----------------------|
| <i>C</i> | - Pentru cerc.       |
| <i>P</i> | - Pentru pătrat.     |
| <i>D</i> | - Pentru dreptunghi. |

## PROGRAMUL BX40

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct {
    int tip;
    union {
        double raza;
        double lp;
        double ld[2];
        double lt[3];
    } fig;
} FIG;

#define EROARE -1
#define CERC 0
#define PATRAT 1
#define DREPTUNGHI 2
#define TRIUNGHI 3

#include "bx38.cpp" /* arie */
#include "bx39.cpp" /* citfig */

main() /* - citește o literă mare care definește o figură geometrică,
        apoi citește elementele figurii respective;
        - calculează și afișează aria acelei figuri. */
{
    char t[255];
    char er[]="s-a tastat EOF\n";
    char lit[2];
    FIG f;
    double a;

    for( ; ; ) {
        printf("tastati una din literele mari:\n");
        printf("C\nD\nP\nT\n");
        if(gets(t) == 0 ){
            printf(er);
            exit(1);
        }
    }
}

```



```

    }
    sscanf(t,"%ls",lit);
    switch(lit[0]) {
        case 'C': /* cerc */
            f.tip = CERC;
            break;
        case 'D': /* dreptunghi */
            f.tip = DREPTUNGHI;
            break;
        case 'P': /* patrat */
            f.tip = PATRAT;
            break;
        case 'T': /* triunghi */
            f.tip = TRIUNGHI;
            break;
        default: /* eroare */
            printf("nu s-a tastat una din literele\
                mari C,D,P sau T\n");
            f.tip = EROARE;
    }
    if(f.tip != EROARE)
        break;
} /* sfirsit for */

/* citeste elementele figurii */
if(citfig(&f) == 0 ) {
    printf(er);
    exit(1);
}

/* calculeaza si afiseaza aria figurii */
if((a=aria(&f)) == 0 )
    exit(1);
printf("aria figurii= %g\n", a );
}

```

10.41 Fie tipul utilizator introdus prin următoarea declarație:

```

typedef struct d_c {
    int zz;
    int tipluna;
    union {
        int nll;

```

```

        char sluna[11];
    } luna;
    int an;
} D_C ;

```

unde:

*tipluna* - Poate avea una din valorile:

INTREG

sau

SIR

Dacă *tipluna* are valoarea INTREG, atunci luna se exprimă prin numărul ei care se păstrează ca valoare a variabilei *nll*. În caz contrar, luna este păstrată prin denumirea ei în tabloul *sluna*.

Funcția de față are ca parametru un pointer spre o dată de tip *D\_C* și returnează numărul lunii calendaristice. La eroare, returnează valoarea 0.

În cazul în care luna calendaristică se dă prin denumirea ei, se utilizează tabloul de pointeri *tpdl* definit în exercițiul 8.19.

## FUNCȚIA BX41

```

int nrluna(D_C *p) /* returneaza numarul lunii sau 0 la eroare */
{
    int i;
    static char *tpdl[] = {"",
        "ianuarie",
        "februarie",
        "martie",
        "aprilie",
        "mai",
        "iunie",
        "iulie",
        "august",
        "septembrie",
        "octombrie",
        "noiembrie",
        "decembrie"
    };

    if(p->tipluna == INTREG)
        /* luna se da prin numarul ei */
        return p->luna.nll < 1 ||

```

```

    p->luna.nll > 12 ? 0 : p->luna.nll;

    if(p->tipluna != SIR )
/* tipluna nu are o valoare corecta */
        return 0;

/* se cauta denumirea lunii in tabloul tpdl */
    for( i=1; i < 13; i++)
        if(strcmp(p->luna.sluna,tpdl[i]) == 0 )
            return i;
    return 0;
}

```

- 10.42 Funcția din acest exercițiu are ca parametru un pointer spre o dată de tip *D\_C*, validează data calendaristică definită de pointerul respectiv și returnează numărul lunii din data respectivă. În cazul în care data este eronată, funcția returnează valoarea zero.

Funcția validează data calendaristică apelînd funcția *v\_calend* definită în exercițiul 6.5.

## FUNCȚIA BX42

```

int vnrluna(D_C *p)
/* - valideaza data calendaristica spre care pointeaza p;
   - returneaza numarul lunii calendaristice si 0 la eroare. */
{
    int nr;

    if((nr=nrluna(p)) == 0)
        return 0; /* luna eronata */
    if(v_calend(p->zz,nr,p->an))
        return nr; /* data calendaristica valida */
    return 0; /* data calendaristica eronata */
}

```

- 10.43 Să se scrie un program care afișează diferența, în zile, dintre două date calendaristice. Una dintre cele două date se dă sub formă de argument în linia de comandă, iar cealaltă se citește de la tastatură.

Notăm cu *d1* data din linia de comandă și cu *d2* data care se tastează. Programul afișează diferența *d2 - d1*, în valoare absolută.

Data *d1* are formatul:

zi luna an

unde:

- zi - Este un întreg de 1-2 cifre.
- luna - Este denumirea lunii calendaristice.
- an - Este întreg de 4 cifre și reprezintă anul din intervalul [1600,4900].

Data *d2* are un format similar cu deosebirea că luna calendaristică se dă prin numărul ei.

Se pot tasta mai multe date *d2* la o aceeași execuție a programului.

### PROGRAMUL BX43

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    int zz;
    int tipluna;
    union {
        int nll;
        char sluna[11];
    } luna;
    int an;
} D_C;

#define INTREG 1
#define SIR 2
#define EROARE -1

#include "bv15.cpp" /*v_calend */
#include "bv16.cpp" /*zi_din_an */
#include "bx41.cpp" /*nrluna */
#include "bx42.cpp" /*vnrluna */
#include "bv1112.cpp" /*pent_int */
#include "bv1113.cpp" /*pent_int_lim */

int nrzile[] = { 0,31,28,31,30,31,30,
                 31,31,30,31,30,31};
```

```

main(int argc, char *argv[])
/* afiseaza numarul de zile dintre doua date calendaristice valide */
{
    char er[]="s-a tastat EOF\n";
    D_C d1,d2;
    int n,n1,n2,t,min,max;

/* pastreaza data calendaristica din linia de comanda in structura d1 */
    if(argc != 4) {
        printf("numar argumente= %d este eronat\n",
            argc);
        exit(1);
    }
    if(sscanf(argv[1],"%d",&d1.zz) != 1) {
        printf("ziua din linia de comanda eronata\n");
        exit(1);
    }
    if(sscanf(argv[2],"%10s",d1.luna.sluna) != 1) {
        printf("luna din linia de comanda eronata\n");
        exit(1);
    }
    if(sscanf(argv[3],"%d",&d1.an) != 1) {
        printf("anul din linia de comanda eronat\n");
        exit(1);
    }
    d1.tipluna = SIR;

/* valideaza data din d1 */
    if((n=vnrluna(&d1)) == 0) {
        printf("data calendaristica din linia de\
            comanda eronata\n");
        printf("zi: %d\tluna: %s\tanul: %d\n",d1.zz,
            d1.luna.sluna, d1.an);
        exit(1);
    }

/* numar zile din an */
    n1=zi_din_an(d1.zz,n,d1.an);

    for( ; ; ) {
        * citeste o data calendaristica de la tastatura si o pastreaza
        in structura d2 */
        if(poit_int_lim("ziua ",1,31,&d2.zz) == 0)

```

```

        exit(0); /* s-a tastat EOF */
    if(pcit_int_lim("numarul lunii calendaristice"
        ,1,12, &d2.luna.n11) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("anul ",1600,4900,&d2.an)==0 )
    {
        printf(er);
        exit(1);
    }
    d2.tipluna = INTREG;

/* valideaza data calendaristica din d2 */
    if(vnrluna(&d2) == 0) {
        printf("data calendaristica tastata este\
            eronata\n");
        printf("z1: %d\tluna: %d\tanul: %d\n",
            d2.zz,d2.luna.n11,d2.an);
        continue;
    }

/* numar zile din an */
    n2=z1_din_an(d2.zz,d2.luna.n11,d2.an);

/* determina relatia dintre ani */
    if(d1.an < d2.an) {
        min=d1.an;
        max=d2.an;
    }
    else {
        min = d2.an;
        max = d1.an;
    }

/* insumeaza diferenta dintre ani in zile */
    for(t=0;min < max; min++)
        t += 365 +(min%4==0 && min%100 ||
            min%400 == 0);
    if(min == d1.an)
        t += n1 - n2; /* t=d1-d2 */
    else
        t += n2- n1; /* t= d2-d1 */

```

```

    if(t < 0 )
        t = -t;

    printf("d1= %d/%d/%d\\td2= %d/%d/%d\\n",d1.zz,
        n,d1.an,d2.zz,d2.luna,n11,d2.an);
    printf("abs(d2 - d1) = %d\\n",t);
}
}

```

## 10.5. Cîmp

Limbajul C permite definirea și prelucrarea datelor pe *biți*. Utilizarea lor poate conduce la economisirea de memorie. Într-adevăr, adesea avem nevoie de date care au numai două valori, zero sau unu. O astfel de dată poate fi păstrată pe un singur bit. De aceea, pentru astfel de date, nu se justifică să alocăm un octet sau chiar doi. În general, nu este util ca date de valori mici să fie păstrate pe octeți sau pe 16 biți, mai ales atunci cînd aceste date sînt în număr mare. În acest scop, limbajul C oferă posibilitatea de a declara date care să se alocate pe biți.

Un șir de biți adiacenți formează un *cîmp*. Un cîmp trebuie să se poată păstra într-un cuvînt calculator.

Mai multe cîmpuri pot fi păstrate într-un același cuvînt calculator.

Cîmpurile se grupează formînd o structură. O astfel de structură se declară ca o structură obișnuită care are ca și componente cîmpuri:

```

struct nume {
    cîmp_1;
    cîmp_2;
    ...
    cîmp_n;
} nume1,nume2,...,numem;

```

Un cîmp se declară astfel:

```
tip nume_cîmp: lungime_in_biți
```

sau

```
tip: lungime_in_biți
```

De obicei, *tip* este cuvîntul cheie *unsigned*, ceea ce înseamnă că șirul de biți din cîmpul respectiv se interpretează ca fiind un întreg fără semn. Alte posibilități pentru *tip* sînt:

- int;
- unsigned char;

și

- char.

Cîmpurile se alocă de la biți de ordin inferior al cuvîntului spre cei de ordin superior.

Cîmpurile cu semn se utilizează pentru a păstra întregi de valori mici prin complement față de doi. De aceea, în acest caz, bitul cel mai semnificativ al cîmpului este bit semn.

Dacă un cîmp nu se poate alocă în cuvîntul curent, el se va alocă în cuvîntul următor.

Un cîmp fără *nume* nu se poate referi. El definește o zonă neutilizată dintr-un cuvînt.

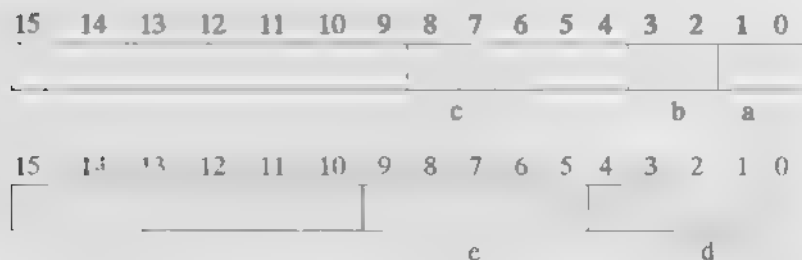
Lungimea în biți poate fi egală cu zero. În acest caz, data următoare se alocă în cuvîntul următor.

Cîmpurile se pot referi folosind aceleași convenții ca și în cazul structurilor obișnuite.

**Exemplu:**

```
struct {
    unsigned a:2;
    int      b:2;
    unsigned i:3;
    unsigned c:2;
    unsigned i:0;
    int      d:5;
    unsigned e:5;
} x,y;
```

Pentru *x* se alocă două cuvinte, astfel:





- $x.a = 1$  - Atribuire cîmpului  $a$  al datei  $x$  valoarea 1, deci bitul 0 devine 1, iar bitul 1 ia valoarea 0.
- $x.b = -1$  - Atribuire cîmpului  $b$ , al datei  $x$ , valoarea -1.
- Aceasta înseamnă că ambii biți ai lui  $b$  se fac egali cu 1 (11 este reprezentarea lui -1 pe 2 biți prin complement față de 2).

Nu se pot defini tablouri de cîmpuri. De asemenea, operatorul adresă (& unar) nu se poate aplica la un cîmp.

Datele pe biți conduc la programe care, de obicei, nu sînt portabile sau au o portabilitate redusă. De aceea, se recomandă utilizarea lor cu precauție.

De asemenea, datele pe biți necesită instrucțiuni suplimentare (deplasări, setări și sau mascări de biți etc) față de cazul cînd sînt păstrate în mod obișnuit (ca date de tip *int* sau *char*). De aceea, utilizarea lor se justifică numai atunci cînd alocarea pe biți conduce la o economie substanțială de memorie față de alocarea pe octeți sau pe cuvinte de 16 biți.

#### Observație:

Prelucrarea datelor pe biți se poate realiza și fără a defini cîmpuri de biți, utilizînd operatori logici pe biți. Utilizarea lor poate conduce însă la un efort de programare suplimentar care poate fi destul de mare. De asemenea, utilizarea operatorilor respectivi poate să nu fie făcută optim sau să conducă la folosirea unor expresii eronate.

Aceasta nu înseamnă că trebuie să renunțăm la utilizarea operatorilor logici pe biți. Există situații cînd utilizarea lor permite scrierea unor programe mai performante decît dacă se utilizează, în aceleași scopuri, cîmpuri de biți.

#### Exerciții:

10.44 Să se scrie o funcție care stabilește dacă o relație binară este o relație de echivalență.

Fie  $A$  și  $B$  două mulțimi (disjuncte sau nu, care pot și coincide).

Prin  $A \times B$  se notează mulțimea perechilor ordonate de forma:

$\langle a, b \rangle$

unde:

- $a$  - Este un element al mulțimii  $A$ , iar  $b$  un element al mulțimii  $B$ .

Mulțimea  $A \times B$  se numește *produsul cartezian* al mulțimilor  $A$  și  $B$ .

O submulțime a produsului cartezian  $A \times B$  se numește *relație binară*.

Dacă  $A = B$ , atunci o submulțime a produsului cartezian  $A \times A$  definește o relație binară pe mulțimea  $A$ .

Fie  $R$  o relație binară, adică o submulțime a produsului cartezian  $A \times B$ . Dacă perechea:

$$\langle a, b \rangle$$

aparține lui  $R$ , atunci se spune că  $a$  și  $b$  se află în relația  $R$ . Notăm acest lucru prin:

$$(1) aRb$$

și se obișnuiește să se spună că expresia (1) este *adevărată*. Dacă  $a$  și  $b$  nu sînt în relația  $R$ , atunci spunem că expresia (1) este *falsă*.

**Exemple:**

1. Dacă  $A = B = \mathbb{N}$ , unde prin  $\mathbb{N}$  am notat mulțimea numerelor naturale, atunci sînt definite o serie de relații binare pe mulțimea  $\mathbb{N}$ . De exemplu, relația *mai mic* este definită pe mulțimea  $\mathbb{N}$  și ea se notează prin simbolul  $<$ .

De exemplu, perechea:

$$\langle 3, 7 \rangle$$

aparține relației mai mic ( $<$ ) și aceasta înseamnă că expresia:

$$3 < 7$$

este adevărată. În schimb expresia:

$$7 < 3$$

este falsă.

2. Fie mulțimile:

$$A = \{1, 2, 3\} \text{ și } B = \{2, 5, 7, 8\}$$

Atunci:

$$A \times B = \{ \langle 1, 2 \rangle, \langle 1, 5 \rangle, \langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 2, 2 \rangle, \langle 2, 5 \rangle, \langle 2, 7 \rangle, \langle 2, 8 \rangle, \langle 3, 2 \rangle, \langle 3, 5 \rangle, \langle 3, 7 \rangle, \langle 3, 8 \rangle \}$$

Relația  $R$  este submulțimea perechilor produsului cartezian cu proprietatea că primul element divide pe al doilea.

Atunci:

$$R = \{ \langle 1, 2 \rangle, \langle 1, 5 \rangle, \langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 2, 2 \rangle, \langle 2, 8 \rangle \}$$

Dacă notăm cu  $R_1$  submulțimea perechilor produsului  $A \times B$  cu proprietatea că primul element este multiplul celui de-al doilea, atunci relația  $R_1$  este o relație vidă.

Fie  $R$  o relație definită pe mulțimea  $A$ .

Dacă

$aRa$

este o expresie adevărată pentru orice  $a$  din  $A$ , atunci se spune că relația  $R$  este *reflexivă*.

Relația  $<$  definită pe mulțimea numerelor naturale nu este o relație reflexivă. Pe aceeași mulțime se poate defini relația de egalitate  $=$ .

Perechea:

$\langle a, a \rangle$

este în relația  $=$ , oricare ar fi  $a$  un număr natural, deci relația  $=$  este reflexivă.

Relația  $R$  definită pe mulțimea  $A$  se spune că este *simetrică* dacă din faptul că expresia:

$aRb$

este adevărată, rezultă că și expresia:

$bRa$

este adevărată.

Relația  $<$ , definită pe mulțimea numerelor naturale nu este simetrică. În schimb, relația  $=$  este simetrică.

În sfârșit, relația  $R$  este *tranzitivă* dacă din faptul că expresiile:

$aRb$  și  $bRc$

sînt adevărate, rezultă că și expresia:

$aRc$

este adevărată.

Relațiile  $<$  și  $=$  definite pe mulțimea numerelor naturale  $N$  sînt tranzitive.

Dacă expresiile:

$a < b$  și  $b < c$

sînt adevărate, rezultă că și expresia:

$$a < c$$

este adevărată oricare ar fi  $a, b, c$  numere naturale.

O relație  $R$  definită pe mulțimea  $A$ , care este reflexivă, simetrică și tranzitivă, se numește relație de echivalență.

Relația  $=$  este un exemplu simplu de relație de echivalență.

Relația  $<$  nu este o relație de echivalență nefiind simetrică și nici reflexivă.

Relația  $\leq$  (mai mic egal) este reflexivă dar nu și simetrică, deci nu este o relație de echivalență.

Relația  $\neq$  (diferit) este o relație simetrică dar nu este reflexivă.

Un exemplu nebanal de relație de echivalență este relația de asemănare definită pe mulțimea triunghiurilor plane. Această relație se notează cu simbolul  $\sim$ .

Dacă  $a$  și  $b$  sînt două triunghiuri plane asemenea, atunci din faptul că  $a$  este asemenea cu  $b$ , rezultă că și  $b$  este asemenea cu  $a$ . Deci, relația de asemănare este simetrică. Ea este și reflexivă, deoarece un triunghi este totdeauna asemenea cu el însuși.

În sfîrșit, dacă  $a, b, c$  sînt trei triunghiuri și au loc relațiile:

$$a \sim b \text{ și } b \sim c,$$

atunci are loc și relația:

$$a \sim c.$$

În acest exemplu, precum și în continuare, vom avea în vedere numai relații cu un număr finit de elemente. Relațiile de acest fel se pot reprezenta prin matrici.

Dacă relația  $R$  este o submulțime a produsului cartezian  $A \times B$ , atunci reprezentăm relația respectivă prin matricea  $a$  astfel:

- Fiecărui element din mulțimea  $A$   $i$  se pune în corespondență o linie în matricea  $a$ .
- Fiecărui element din mulțimea  $B$   $j$  se pune în corespondență o coloană în matricea  $a$ .
- Fie  $a_i$  elementul mulțimii  $A$  căruia îi corespunde linia  $a$   $i$ -a din matricea  $a$  și  $b_j$  elementul mulțimii  $B$  căruia îi corespunde coloana  $a$   $j$ -a din matricea  $a$ . Atunci:

$$a[i][j] = 1, \text{ dacă } a_i R b_j$$

și

$$a[i][j] = 0, \text{ altfel.}$$

**Exemplu:**

Să considerăm exemplul 2 de mai sus. Matricea  $a$  conține 3 linii și 4 coloane. Relația  $R$  se reprezintă prin următoarea matrice:

		2	5	7	8
	1	1	1	1	1
A	2	1	0	0	1
	3	0	0	0	0



În cazul în care  $A = B$ , matricea este pătratică. Relațiile de echivalență se definesc pentru astfel de cazuri. Proprietatea de reflexivitate se exprimă simplu și anume, matricea conține 1 pe diagonala principală.

Într-adevăr, relația  $R$  este reflexivă dacă:

$$a_i R a_i$$

este adevărată, adică dacă

$$a[i][i] = 1, \text{ pentru toate valorile lui } i.$$

Relația  $R$  este simetrică dacă matricea de reprezentare a ei este simetrică.

Într-adevăr, dacă  $R$  este simetrică, atunci din faptul că expresia:

$$a_i R a_j$$

este adevărată, rezultă că și expresia.

$$a_j R a_i$$

este adevărată. Aceasta înseamnă că:

$$a[i][j] = a[j][i] = 1$$

iar dacă:

$$a[i][j] = 0, \text{ atunci } a[j][i] = 0$$

deci:

$$a[i][j] = a[j][i]$$

Pentru a verifica tranzitivitatea relației, să considerăm definiția tranzitivității:

$R$  este tranzitivă dacă din faptul că expresiile:

$$a_i R a_j \text{ și } a_j R a_k$$

sînt adevărate, rezultă că și expresia:

$a_i R a_k$

este adevărată.

Deci, este necesar să se verifice că dacă

$$a[i][j] = a[j][k] = 1$$

atunci și

$$a[i][k] = 1, \text{ pentru toate valorile lui } i, j \text{ și } k.$$

Dacă există cel puțin un element

$$a[i][k] = 0$$

în timp ce

$$a[i][j] = a[j][k] = 1$$

pentru cel puțin un  $j$ , atunci relația nu este tranzitivă.

Elementele matricei de reprezentare a unei relații au numai două valori: 0 sau 1.

De aceea, o astfel de matrice se poate păstra pe biți. Aceasta permite realizarea unei economii de memorie importantă față de cazurile în care matricea se păstrează pe octeți sau cuvinte de 16 biți. De exemplu, dacă se utilizează o matrice de ordinul 100, atunci necesarul de memorie va fi:

$$100 \cdot 100 \text{ biți} = 10000/8 \text{ octeți} = 1250 \text{ octeți},$$

față de 10000 de octeți, cînd matricea s-ar păstra pe octeți.

Avînd în vedere acest fapt, funcția de față utilizează o matrice cu elemente păstrate pe biți, pentru a reprezenta relația de analizat.

Ea returnează valoarea 1 dacă relația este de echivalență și zero în caz contrar.

Matricea relației se păstrează într-un tablou unidimensional de tip *char* și fiecare element al tabloului păstrează 8 elemente consecutive din matricea respectivă.

Fie  $a$  matricea relației, care este o matrice pătratică de ordinul  $n$ .

Notăm cu  $ta$  tabloul unidimensional care păstrează elementele matricei  $a$  pe biți. Problema care ne interesează este accesul la elementul  $a[i][j]$  al matricei  $a$ , reprezentată prin tabloul  $ta$  ( $i = 0, 1, \dots, n-1$  și  $j = 0, 1, \dots, n-1$ ).

Dacă elementele matricei  $a$  s-ar păstra pe octeți, atunci folosind relația de liniarizare (vezi exercițiul 5.1), acest element ar avea indicele:

$$k = i * n + j$$

deci

$a[i][j]$  ar fi elementul  $ta[k = i * n + j]$ .

Deoarece fiecare element al tabloului  $ta$  păstrează 8 elemente din matricea  $a$ , rezultă că valoarea lui  $k$ , de mai sus, trebuie micșorată de 8 ori.

Avem:

$a[0][0], a[0][1], \dots, a[0][7]$ , se păstrează în  $ta[0]$ ,  
 $a[0][8], a[0][9], \dots, a[0][15]$ , se păstrează în  $ta[1]$

...

În general, elementul  $a[i][j]$  se păstrează în octetul de indice  $(i * n + j) / 8$ , adică este un bit al elementului:

$ta[(i * n + j) / 8]$

Poziția bitului se determină astfel:

- dacă  $(i * n + j) \% 8$  este zero,  
atunci este bitul cel mai semnificativ (bitul 7)
- dacă  $(i * n + j) \% 8 = 1$ ,  
atunci este bitul 6;
- în general, dacă  $(i * n + j) \% 8 = r$ ,  
atunci este bitul  $7 - r$ .

## FUNCȚIA BX44

```
unsigned belem(char [], int, int, int);
```

```
int relechiv(char ta[], int n)
```

```
/* returneaza 1 daca relatia pastrata in tabloul ta pe biti este o relatie de echivalenta si zero in caz contrar */
```

```
{
    int i, j, k;
```

```
/* reflexivitate */
```

```
for(i=0; i < n ; i++)
    if(belem(ta, n, i, i) == 0 )
```

```

        return 0; /* relatia nu este reflexiva */

    for( i=0; i < n; i++ )
        for( j=0; j<n; j++ )
            if( belem( ta, n, i, j) != belem( ta, n, j, i) )
                return 0; /* relatia nu este simetrica */

    for( i=0; i < n; i++ )
        for( j=0; j < n; j++ )
            if( belem( ta, n, i, j) )
                for( k=0; k < n; k++ )
                    if( belem( ta, n, j, k) )
                        if( belem( ta, n, i, k) == 0 )
                            return 0;

/* relatia este tranzitiva */
    return 1;
}

unsigned belem( char ta[], int n, int i, int j )
/* - returneaza valoarea elementului a[i][j];
   - elementele matricei patratice a, de ordinul n, sint pastrate pe biti in
   tabloul ta, folosind relatia de liniarizare. */
{
    return ta[ (i*n+j)/8 ] >> 7 - (i*n+j)%8 & 1;
}

```

### Observație:

Funcția *belem* returnează valoarea elementului  $a[i][j]$ . Așa cum s-a indicat mai sus, elementul respectiv este păstrat într-un bit al elementului:

$ta[(i*n+j)/8]$ .

Selectarea acestui bit se face deplasând spre dreapta valoarea acestui element cu:

$7 - (i*n+j)\%8$

poziții binare (prin aceasta bitul în cauză devine cel mai puțin semnificativ) iar apoi se face un *și logic pe biți* dintre rezultatul acestei deplasări și 1 (în felul acesta se anulează toți biții excepiind ultimul, care este chiar valoarea lui  $a[i][j]$  ).

Această funcție se poate realiza folosind cimpuri de biți. În acest scop, se folosește o reuniune cu două componente. Una este o structură care



definește fiecare bit al unui octet, iar cealaltă este o dată de tip *char*. Elementul `ta[(i*n+j)/8]` se atribuie datei de tip *char* și apoi se selectează bitul dorit prin acces direct la el.

```
unsigned belem(char ta[], int n, int i, int j)
/* returnează elementul a[i][j] pastrat pe biti in tabloul ta */
{
    union {
        struct {
            unsigned b0:1;
            unsigned b1:1;
            unsigned b2:1;
            unsigned b3:1;
            unsigned b4:1;
            unsigned b5:1;
            unsigned b6:1;
            unsigned b7:1;
        } b;
        char x;
    } octet;

    /* pastreaza octetul ce contine bitul de selectat in zona
    alocata reuniunii */
    octet.x = ta[(i*n+j)/8];

    /* selecteaza bitul de ordin 7-(i*n+j)%8 */
    switch ( 7-(i*n+j)%8 ) {
        case 0:
            return octet.b.b0;
        case 1:
            return octet.b.b1;
        case 2:
            return octet.b.b2;
        case 3:
            return octet.b.b3;
        case 4:
            return octet.b.b4;
        case 5:
            return octet.b.b5;
        case 6:
            return octet.b.b6;
        case 7:
            return octet.b.b7;
    }
}
```

}

10.45 Să se scrie o funcție care calculează compunerea a două relații.

Fie  $P$  și  $Q$  două relații reprezentate prin matricele  $MP$  și  $MQ$ .

Operatorul de compunere a două relații îl notăm prin caracterul punct.

Fie  $R$  compunerea celor două relații  $P$  și  $Q$ :

$$R = P.Q$$

Operatorul de compunere a două relații se definește ca mai jos.

Fie  $P$  o submulțime a produsului cartezian:

$$A \times B$$

și  $Q$  o submulțime a produsului cartezian:

$$B \times C$$

Atunci  $R$  este o submulțime a produsului cartezian:

$$A \times C$$

care se definește astfel:

perechea

$$\langle a, c \rangle$$

aparține relației  $R$ , dacă există un element  $b$ , în mulțimea  $B$ , așa încît perechea:

$$\langle a, b \rangle$$

aparține relației  $P$ , iar perechea:

$$\langle b, c \rangle$$

aparține relației  $Q$ .

Deci, dacă expresiile:

$$aPb \text{ și } bQc$$

sînt adevărate, atunci este adevărată și expresia:

$$aRc$$

Fie  $MR$  matricea care reprezintă relația  $R$ . Dacă  $MP$  este o matrice de ordinul  $m \times n$ , iar  $MQ$  o matrice de ordinul  $n \times s$ , atunci matricea  $MR$  este de ordinul  $m \times s$ .

Matricea relației  $MX$  ( $X = P, Q$  sau  $R$ ) este păstrată pe biți în tabloul

unidimensional tmx ( $x = p, q$  sau  $r$ ).

## FUNCȚIA BX45

```
unsigned belem(char [],int,int,int);
```

```
void sbelem(char [],int,int,int);
```

```
void prodrel(char tmp[],char tmq[],char tmr[],  
              int m,int n,int s)
```

```
/* - calculeaza relatia:
```

$R = P.Q$

- relatia P se reprezinta prin matricea MP;

- relatia Q se reprezinta prin matricea MQ;

- relatia R se reprezinta prin matricea MR;

- matricea MP este de ordinul  $m \times n$  si se pastreaza pe biti in tabloul tmp;

- matricea MQ este de ordinul  $n \times s$  si se pastreaza pe biti in tabloul tmq;

- matricea MR este de ordinul  $m \times s$  si se pastreaza pe biti in tabloul tmr. \*/

```
{  
  int i,j,k;
```

```
/* se anuleaza toate elementele matricei MR */
```

```
  for(i=0; i<=m*s/8; i++)  
    tmr[i] = 0;
```

```
/* se seteaza elementele matricei MR */
```

```
  for(j=0; j<n; j++)  
    for(i=0; i<m; i++)  
      if(belem(tmp,n,i,j))
```

```
/* MP[i][j] = 1 */
```

```
      for(k=0; k<s; k++)  
        if(belem(tmq,s,j,k))
```

```
/* MQ[j][k] = 1 */
```

```
/* elementul MR[i][k] se seteaza la valoarea 1 */  
      sbelem(tmr,s,i,k);
```

```
}
```

```

void sbelem(char ta[],int n, int i,int j)
/* - seteaza la 1 valoarea elementului a[i][j];
   - matricea a este pastrata in tabloul ta pe biti folosind relatia de
     liniarizare;
   - matricea a are n coloane. */
{
    ta[(i*n+j)/8] |= 1<< 7 -(i*n+j)%8;
}

```

### Observație:

Expresia:

$$(1) 1 < 7 - (i*n+j) \% 8$$

deplasează pe unu spre stînga cu un număr de poziții binare egal cu valoarea expresiei:

$$(2) 7 - (i*n+j) \% 8$$

Se observă că dacă  $i*n+j$  este multiplu de 8, atunci expresia (2) are valoarea 7 și deci unu se deplasează cu 7 poziții binare spre stînga. Se obține:

10000000

deci, bitul cel mai semnificativ al octetului rezultat, devine egal cu 1.

Dacă restul împărțirii lui  $i*n+j$  la 8 este 1, atunci expresia (2) are valoarea 6, iar expresia (1) generează valoarea:

1000000

Deci, în acest caz, bitul de ordinul 6 al rezultatului, devine egal cu 1.

În mod analog, dacă  $(i*n+j) \% 8 = 2$ , bitul de ordinul 5 al rezultatului devine egal cu 1, și așa mai departe.

Dacă  $(i*n+j) \% 8 = 7$ , atunci expresia (2) are valoarea zero, iar expresia (1) are valoarea 1. Deci, bitul de ordinul zero al rezultatului va deveni egal cu 1.

10.46 Să se scrie o funcție care citește indicii elementelor  $a[i][j]$  diferite de zero ale matricei  $a$  care reprezintă o relație binară. Matricea este de ordinul  $m*n$  și elementele se păstrează pe biti, în tabloul  $ta$  unidimensional, folosind relația de liniarizare.

La început funcția citește valorile lui  $m$  și  $n$ . Apoi se citesc indicii  $i$  și  $j$  pentru fiecare element nenul al matricei  $a$ .

## FUNCȚIA BX46

```

void citirel(char ta[], int max, int *m, int *n)
/* - citește pe m și n;
   - citește indicii elementelor nenule ale matricei a ce reprezintă
      o relație binară;
   - păstrează matricea pe biti în tabloul ta folosind relația de liniarizare.
*/
{
    int i, j;

    do { /* citește pe m și n */
        *m = *n = 0;
        if (pcit_int_lim("număr linii m = ",
                        1, max, m) == 0) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if (pcit_int_lim("număr coloane n = ", 1, max,
                        n) == 0) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if (*m * *n < max)
            break;
        printf("produsul m*n= %d\n", *m * *n);
        printf("depășește valoarea maximă= %d\n", max);
    } while (1);
    for (i = 0; i <= *m * *n / 8; i++)
        ta[i] = 0;

    /* citește indicii elementelor nenule și setează bitii corespunzători în
       tabloul ta */
    for ( ; ; ) {
        if (pcit_int_lim("linia i = ", 1, *m, &i) == 0)
            return;
        if (pcit_int_lim("coloana j = ", 1, *n, &j) == 0)
            return;

        /* setează elementul a[i-1][j-1] */
        ta[((i-1) * *n + j-1) / 8] |= 1 <<
            7 - ((i-1) * *n + j-1) % 8;
    }
}

```

10.47 Să se scrie un program care calculează și afișează compunerea a două relații definite pe mulțimea A, apoi stabilește dacă relațiile respective sînt de echivalență.

Relațiile se introduc folosind funcția *citrel* definită în exercițiul precedent.

Apoi, se apelează funcția *prodrel* care compune relațiile respective. În final, se stabilește dacă cele trei relații sînt de echivalență.

## PROGRAMUL BX47

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "bviii2.cpp" /* pcit_int    */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bx44.cpp"   /* relechiv  */
#include "bx45.cpp"   /* prodrel   */
#include "bx46.cpp"   /* citrel    */

void afisrel(char *, char [],int );

#define MAX 128

main() /*- calculeaza si afiseaza compunerea a doua relatii;
      - stabileste daca relatule sînt de echivalenta. */
{
    char a[MAX*(MAX/8)], b[MAX*(MAX/8)],
          ab[MAX*(MAX/8)];
    int m,n,p,q;

    /* defineste relatia reprezentata prin tabloul a */
    citrel(a,MAX*MAX,&m,&n);

    /* defineste relatia reprezentata prin tabloul b */
    citrel(b,MAX*MAX,&p,&q);
    if(1(m==n&& n==p && p==q )) {
        printf("se cere ca relatiile sa fie\
              reprezentate prin matrice patratice\n");
        exit(1);
    }
}
```

```

/* se calculeaza compunerea relatiilor a si b:
   ab = a.b */
prodrel(a,b,ab,m,m,m);

/* afiseaza relatia reprezentata prin a */
afisrel("relatia a ", a,m);

/* afiseaza relatia reprezentata prin b */
afisrel("relatia b ", b , m );

/* afiseaza compunerea relatilor a si b */
afisrel("relatia ab= a.b ",ab, m);

/* stabileste care dintre relatii este de echivalenta */
if(relechiv(a,m))
    printf("relatia a este de echivalenta\n");
else
    printf("relatia a nu este de echivalenta\n");
if(relechiv(b,m))
    printf("relatia b este de echivalenta\n");
else
    printf("relatia b nu este de\
    echivalenta\n");
if(relechiv(ab,m))
    printf("relatia ab este de\
    echivalenta\n");
else
    printf("relatia ab nu este de\
    echivalenta\n");
} /* sfirsit main */

void afisrel(char *p, char ta[],int n)
/* afiseaza relatia pastrata pe biti prin linarizare in tabloul ta */
{
    int i,j,k,m;.

/* afiseaza un anctet */
    printf("\n\n\t\t%s\n\n",p);
    for(i=0; i< n; i++) {
        printf("linia: %d\n", i+1);
        k=1,m=0 ;
        for(j=0; j<n; j++,k++) {
            printf("%d ",

```

```

        ta[(i+n+j)/8]>> 7-(i+n+j)%8&1);
if( k == 30 ) {
    printf("\n");
    k = 0;
    m++;
    if(m%20 == 0 ) {
        printf("actionati o tasta pentru a\
        continua\n");
        getch();
    }
}
}
printf("actionati o tasta pentru a\
        continua\n");
getch();
}
}

```

## 10.6. Tipul enumerare

Tipul *enumerare* permite programatorului să folosească nume sugestive pentru valori numerice. De exemplu, în locul numărului unei luni calendaristice este mai sugestiv să folosim denumirea lunii respective sau eventual o prescurtare:

*ian*                    - Pentru ianuarie în locul cifrei 1.  
*feb*                    - Pentru februarie în locul cifrei 2.

și așa mai departe.

Un alt exemplu se referă la posibilitatea de a utiliza cuvintele FALS și ADEVARAT pentru valorile 0 respectiv 1. În felul acesta se obține o mai mare claritate în programele sursă, deoarece valorile numerice sînt înlocuite prin sensurile atribuite lor într-un anumit context.

În acest scop se utilizează tipul *enumerare*. Un astfel de tip se declară printr-un format asemănător cu cel utilizat în cadrul structurilor. Un prim format general este:

```
enum nume { nume0,nume1,nume2,...,numek } d1,d2,...,dn;
```

unde:

*nume*                    - Este numele tipului de enumerare introdus prin această declarație.



*nume0,nume1,...,  
numek*

- Sînt nume care se vor utiliza în continuare în locul valorilor numerice și anume *numei* are valoarea *i*.

*d1,d2,...,dn*

- Sînt date care se declară de tipul *nume*.

- Aceste date sînt similare cu datele de tip *int*.

Ca și în cazul structurilor, în declarația de mai sus nu sînt obligatorii toate elementele. Astfel, poate lipsi nume, dar atunci va fi prezent cel puțin *d1*. De asemenea, poate lipsi în totalitate lista *d1,d2,...,dn*, dar atunci va fi prezent *nume*. În acest caz, se vor defini ulterior date de tip *nume* folosind un format de forma:

**enum** *nume d1,d2,...,dn;*

**Exemple:**

1. **enum** { ileg,ian,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec } luna;

Prin această declarație, numărul lunii poate fi înlocuit prin denumirea prescurtată a lunii respective. De exemplu, o atribuire de forma:

*luna* = 3

se poate înlocui cu una mai sugestivă:

*luna* = mar

deoarece, conform declarației de mai sus, *mar* are valoarea 3.

În mod analog, o expresie de forma:

*luna* == 7

este identică cu expresia:

*luna* == iul

Dacă în locul declarației de mai sus s-ar fi utilizat declarația de tip enumerare:

**enum** *d1* { ileg,ian,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec };

atunci putem declara ulterior data *luna* de tip *d1* astfel:

**enum** *d1* luna;

Data *luna* declarată în acest fel este o dată identică cu data *luna* declarată la început.

2. Fie tipul enumerare *Boolean* declarat astfel:

**enum** Boolean { false, true};

Declarăm data *bisect* de tip Boolean:

```
enum Boolean bisect;
```

Atribuirea:

```
bisect = an%4 == 0 && an%100 || an%400 == 0;
```

atribuie variabilei *bisect* valoarea 1 sau 0, după cum anul definit de variabila *an* este bisect sau nu (se presupune că anul aparține intervalului [1600,4900]).

În continuare se pot folosi expresii de forma:

```
bisect == false
```

sau

```
bisect == true
```

3. Fie *f* o funcție care returnează numai două valori: 0 sau 1. O astfel de funcție se consideră, de obicei, că returnează o valoare booleană. Atunci, antetul ei poate fi definit astfel:

```
enum Boolean f( ... )
```

În continuare se pot folosi expresii de forma:

```
f(...) == false
```

```
f(...) == true
```

La declararea tipurilor enumerare se poate folosi cuvântul cheie *typedef*, ca și în cazul tipurilor utilizator definite prin *struct*. În acest caz vom utiliza o declarație de forma:

```
typedef enum nume { nume0,nume1,...,numek } nume_tip;
```

În continuare, se pot declara mai simplu date de tipul enumerare *nume* folosind:

```
nume_tip d1,d2,...,dn;
```

În acest caz *nume* este opțional și de obicei lipsește. Folosind această convenție, exemplele de mai sus pot fi scrise astfel:

```
typedef enum { ian, feb, mar, apr, mai, iun,  
              iul, aug, sep, oct, nov, dec } DL;
```

```
DL luna;
```

```
typedef enum { false, true } Boolean;
```

```
Boolean bisect;
```

sau antetul funcției *f*:

Boolean *f*(...)

Tipul enumerare poate fi declarat impunind valori altele decât cele care rezultă implicit. În acest caz, *numei* se va înlocui cu:

*numei* = *eci*

unde:

*eci* - Este o expresie constantă.

Numele *numei* va avea ca valoare, valoarea expresiei *eci*. Dacă numelui următor nu i se atribuie o valoare, atunci acesta va avea ca valoare, valoarea lui *numei* mărit cu 1.

Folosind această observație, modificăm tipul DL astfel:

```
typedef enum { ian=1,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec } DL;
```

Menționăm că datele de acest tip nu sînt supuse la controale din partea compilatorului C și de aceea se pot scrie expresii care să nu corespundă scopului pentru care s-au definit datele respective.

De exemplu, fie:

```
DL d1,d2,d3;
```

Expresiile de mai jos nu sînt interpretate ca eronate de compilator:

```
d3 = d1 + d2;
```

```
d3 = d1 * d2;
```

```
d3 = d1/d2; etc.
```

Aceasta, deoarece datele de tip enumerare sînt tratate ca simple date de tip *int*.

## 10.7. Date definite recursiv

Fie declarația:

```
struct nume {  
    declaratii  
};
```

Declarațiile incluse între acolade definesc componentele datelor structurate de tip *nume*, tip definit prin declarația *struct*. Aceste declarații pot defini date de diferite tipuri predefinite sau utilizator, dar diferite de tipul *nume*. În schimb, se pot defini pointeri spre date de tipul *nume*. Deci, o declarație de forma:

```

struct nume {
    declaratii
    struct nume *nume1;
    declaratii
};

```

este corectă, în timp ce declarația:

```

struct nume {
    declaratii
    struct nume nume1;
    declaratii
};

```

este eronată.

Un tip utilizator, se spune că este *direct recursiv* dacă el are cel puțin o componentă care este de tip *pointer spre el însuși*.

**Exemplu:**

```

struct tnod {
    char cuvint[100];
    int nr;
    struct tnod *urm;
} ;

```

Tipul utilizator *tnod* conține 3 componente:

- tabloul *cuvint* de tip *char* de 100 elemente;
- variabila *nr* de tip *int*;
- pointerul *urm* spre tipul *tnod*.

*tnod* este un tip direct recursiv deoarece conține un pointer spre el însuși.

În continuare, putem defini obișnuit date de tipul *tnod*.

Evident, tipurile direct recursive pot fi denumite folosind construcția *typedef*. Tipul de mai sus îl vom numi *TNOD* folosind declarația:

```

typedef struct tnod {
    char cuvint[100];
    int nr;
    struct tnod *urm;
} TNOD;

```

În continuare putem defini date folosind tipul *TNOD*:

```

TNOD nod, *p;

```

O altă posibilitate în definirea tipurilor utilizator este acela în care un tip *t1* conține un pointer spre tipul *t2*, iar acesta, la rândul lui, conține un pointer spre tipul *t1*. În acest caz se spune că tipul *t1* este *indirect recursiv*.

Să observăm că la declararea tipului *t1*, se utilizează o declarație de forma:

```
struct t2 *nt2;
```

Aceasta este posibil dacă tipul *t2* este definit înainte de tipul *t1*. Să presupunem că acest lucru are loc. Dar la declararea tipului *t2* constatăm că acesta conține o declarație de forma:

```
struct t1 *nt1;
```

Aceasta la rândul ei poate fi scrisă dacă înaintea declarației tipului *t2* este prezentă declarația care definește tipul *t1*. Se ajunge în felul acesta la o contradicție. Pentru a o elimina, s-a introdus în limbajul C declarația de tip incompletă.

Aceasta are formatul:

```
struct nume;
```

Această declarație poate fi plasată într-un fișier sursă înainte de a defini tipul *nume* sau în orice fișier sursă în care tipul *nume* nici nu este definit. După declarația de tip incompletă, se pot declara pointeri spre tipul respectiv. În felul acesta, tipurile *t1* și *t2* de mai sus, se vor declara astfel:

```
struct t1; /* - declaratie incompleta pentru tipul t1,  
           - in continuare se poate declara tipul t2 */
```

```
struct t2 {  
    declaratii  
    struct t1 *nt1;  
    declaratii  
};  
struct t1 { /* declaratia completa a lui t1 */  
    declaratii  
    struct t2 *nt2;  
    declaratii  
};
```

Un tip se spune că este *recursiv* dacă el este direct sau indirect recursiv.

O *dată este recursivă* dacă ea este o dată de un tip recursiv.

În general, un tip *t* este *indirect recursiv* dacă există un șir de tipuri *ti*:

*t1, t2, t3, ..., tn*

aşa încît tipul  $ti$  conţine cel puţin o componentă care este pointer spre tipul  $tj$  ( $j=i+1$ ), pentru  $i = 1, 2, \dots, n-1$ , iar  $t1$  şi  $tn$  sînt ambele de tipul  $t$ .

Datele recursive se utilizează într-o serie de aplicaţii bazate pe definirea şi prelucrarea listelor, arborilor, tabelelor de dispersie etc.

Ele se folosesc cu succes în prelucrarea dinamică a datelor.

În multe aplicaţii se utilizează date structurate de un anumit tip, care se repetă de un număr variabil de ori, număr care se precizează la fiecare execuţie a programului.

În astfel de situaţii se poate defini un tablou de structuri a cărui număr de elemente să fie egal cu o valoare maximă precizată de utilizator. Acest mod de lucru nu este cel mai indicat atunci cînd numărul datelor diferă mult de la o execuţie la alta. De asemenea, pot să se ivească situaţii în care acest maxim este greu de evaluat, ca să nu mai vorbim de faptul că el poate fi evaluat greşit. De aceea, în astfel de cazuri este mult mai util să se păstreze datele respective în memoria *heap*, rezervindu-se memorie pentru ele la execuţie, pe măsură ce este nevoie. Prelucrarea datelor respective implică posibilitatea de a trece simplu de la o dată la alta. În cazul tablourilor, aceeaşi se asigură cu ajutorul indicilor. În noua situaţie se poate trece de la o dată păstrată în memoria *heap* la o altă dată de acelaşi tip, dacă se utilizează un pointer spre tipul comun datelor respective şi care să fie componentă a datei respective. Deci, se ajunge la un tip de forma:

```
typedef struct nume {  
    declarat  
    struct nume *urm;  
    declarat  
} nume_tip;
```

În felul acesta se ajunge la necesitatea de a utiliza date recursive.

Gestiunea dinamică a lor mai are încă un avantaj şi anume: astfel de date pot fi eliminate uşor din memoria *heap* cînd nu este nevoie de ele şi zona eliberată în acest fel poate fi realocată altor date de acelaşi tip sau în alte scopuri.

### Exerciţii:

10.48 Să se scrie o funcţie care citeşte un cuvînt şi-l păstrează în memoria *heap*. Prin cuvînt înţelegem o succesiune de litere mici sau mari.

Funcţia returnează adresa de început a zonei din memoria *heap* în care se păstrează cuvîntul citit sau zero la întîlnirea sfîrşitului de fişier.

## FUNCȚIA BX48

```
char *citcuv()
/* - citește un cuvânt și-l păstrează în memoria heap;
   - returnează pointerul spre cuvântul respectiv sau zero la sfârșit
   de fișier. */
{
    int c,i;
    char t[255];
    char *p;

    /* salt peste caractere care nu sînt litere */
    while((c=getchar()) < 'A' || (c > 'Z' &&
        c < 'a') || c > 'z')
        if( c == EOF )
            return 0; /* s-a tastat EOF */

    /* se citește cuvîntul și se păstrează în t */
    i = 0;
    do {
        t[i++] = c;
    } while((c=getchar()) >= 'A' &&
        c <= 'Z' || c >= 'a' && c <= 'z');
    if( c == EOF )
        return 0;
    t[i++] = '\0';

    /* se păstrează cuvîntul în memoria heap */
    if((p = (char *)malloc(i)) == 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }
    strcpy(p,t);
    return p;
}
```

10.49 Să se scrie o funcție care citește cuvintele dintr-un text și determină numărul de apariții al fiecărui cuvînt din textul respectiv.

Această problemă poate fi rezolvată în mai multe moduri. Prezentăm mai jos o metodă simplă, care însă nu este și eficientă. Metode mai eficiente vor fi prezentate în alte capitole.

Înainte de toate să observăm că problema se poate rezolva folosind un tablou de pointeri spre *char*, dacă putem indica o valoare maximă pentru

numărul de cuvinte diferite din text. În acest caz se definește tabloul:

```
char *tp[maxcuv];
```

și un indice *itp* care are ca valoare indicele primului element liber din tablou.

De asemenea, vom utiliza un tablou pentru a număra aparițiile fiecărui cuvânt:

```
int nc[maxcuv];
```

Procesul de calcul se realizează astfel:

1. *itp* = 0.
2. Se apelează funcția *citcuv* pentru a citi cuvântul curent:

```
p = citcuv();
```

Dacă *p* = 0, s-a întâlnit sfârșitul de fișier și procesul de calcul se întrerupe.

Altfel se trece la pasul următor.

3. Se stabilește dacă cuvântul citit (spre care pointează *p*) a fost deja întâlnit în text.

În acest scop se compară cuvintele spre care pointează elementele:

```
tp[0], tp[1], ..., tp[itp-1]
```

cu cel spre care pointează *p*.

În caz afirmativ, se trece la pasul 4, altfel la pasul 5.

4. Dacă cuvântul spre care pointează *p* coincide cu cel spre care pointează *tp[j]*, atunci se incrementează *nc[j]* (numărul de apariții al cuvântului respectiv); apoi se elimină cuvântul, citit la punctul 2, din memoria *heap* și procesul de calcul se reia de la pasul 2.
5. Deoarece cuvântul spre care pointează *p* nu a fost încă întâlnit în textul de intrare, pointerul *p* se păstrează și numărul de apariții al cuvântului respectiv se face egal cu 1:

```
tp[itp] = p;
```

```
nc[itp] = 1;
```

apoi se mărește *itp* cu o unitate:

```
itp = itp + 1;
```

procesul de calcul se reia de la pasul 2.



Funcția utilizează un tablou de structuri de tipul T declarat astfel:

```
typedef struct {  
    char *tp;  
    int nc;  
} T;
```

Ea are un parametru care este un tablou de tipul T. Un alt parametru al ei este *maxcuv*.

Funcția returnează numărul cuvintelor distincte citite (*itp*).

## FUNCȚIA BX49

```
int frecvcuv(T t[], int maxcuv)  
/* - citește un text și determină frecvența de apariție a fiecărui cuvint  
   citit;  
   - se admit cel mult maxcuv cuvinte diferite;  
   - funcția returnează numărul cuvintelor distincte întâlnite în text */  
{  
    char *p;  
    int itp;  
    int i;  
  
    for(itp = 0; itp < maxcuv; ) {  
  
        /* citește cuvântul curent */  
        if(( p = citcuv() ) == 0 )  
  
            /* s-a întâlnit sfârșitul de fișier */  
            return itp;  
  
        /* se stabilește dacă cuvântul citit a fost deja întâlnit */  
        for(i = 0; i < itp; i++)  
            if(strcmp(t[i].tp, p) == 0 )  
                break;  
        if( i < itp ) {  
  
            /* cuvint deja întâlnit */  
            t[i].nc++;  
            free(p);  
        }  
        else { /* cuvint nou */  
            t[itp].tp = p;  
            t[itp++].nc = 1;  
        }  
    }  
}
```

```

    }
}
return itp;
}

```

10.50 Să se scrie un program care afișează frecvența de apariție a cuvintelor unui text.

#### PROGRAMUL 87111

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>

typedef struct {
    char *tp;
    int nc;
} T ;

#include "bx48.cpp"
#include "bx49.cpp"

#define MAXCUV 100

main() /* afiseaza frecventa de aparitie a cuvintelor unui text */
{
    T tab[MAXCUV];
    int ncuv,i;

    ncuv = frecvcuv(tab,MAXCUV);
    for(i=0; i < ncuv; i++) {
        printf("%s\t%d\n",tab[i].tp,tab[i].nc);
        if((i+1)%23 == 0 ) {
            printf("actionati o tasta pentru a\
                continua\n");
            getch();
        }
    }
}

```

10.51 Să se modifice funcția *frecvcuv* definită în exercițiul 10.49, în așa fel

incît în loc de tabloul de tip T să se utilizeze date de tip recursiv.

În acest caz vom folosi tipul recursiv:

```
typedef struct tr {  
    char *tp;  
    int nc;  
    struct tr *urm;  
} TR;
```

Pentru fiecare cuvînt se va păstra, în memoria heap, o dată de tip TR. Pointerului *urm* i se atribuie adresa de început a zonei din memoria heap în care se păstrează cuvîntul următor citit din textul de intrare. El are valoarea zero pentru data corespunzătoare ultimului cuvînt citit.

Pentru a gestiona simplu datele păstrate în memoria heap, va fi util să existe două variabile de tip pointer spre TR, una să poarte spre prima dată păstrată în memoria heap, iar cealaltă spre ultima. Vom numi *prim* și *ultim* aceste variabile.

Funcția realizează pașii de mai jos:

1. *prim* = *ultim* = 0.
2. Se apelează funcția *citcuv* pentru a citi cuvîntul curent:

*p* = *citcuv*();

Dacă *p* = 0, atunci se revine din funcție cu valoarea variabilei *prim*; altfel se trece la pasul următor.

3. Se stabilește dacă cuvîntul citit (spre care pointează *p*) a fost deja întîlnit în text

În acest scop se compară cuvintele spre care pointează pointerul *tp* din datele de tip TR păstrate în memoria heap, cu cel spre care pointează *p*.

În caz afirmativ se incrementează numărătorul cuvîntului respectiv, se eliberează zona ocupată de cuvîntul curent citit și procesul se reia de la pasul 2.

Altfel, se rezervă zonă pentru o dată de tip TR și fie *q* adresa ei de început. Pointerului *urm*, corespunzător ultimei date de tip TR păstrate în memoria heap, i se atribuie valoarea lui *q*

Apoi *nc* = 1 și *urm* = 0 pentru data spre care pointează *q*.

Cum aceasta devine ultima dată păstrată în memoria heap, se face *ultim*=*q* și apoi se revine la pasul 2.

## FUNCȚIA BX51

```
TR *pfrecvcuv()  
/* citește un text și păstrează în memoria heap date de tip TR pentru  
   cuvintele distincte din text, determină frecvența acestora și retur-  
   nează pointerul spre data corespunzătoare primului cuvânt din text */  
{  
    TR *prim, *ultim,*q;  
    char *p;  
  
    prim = ultim = 0;  
    for( ; ; ) {  
  
        /* citește cuvântul curent */  
        if((p = citcuv()) == 0)  
  
            /* s-a întâlnit sfârșitul de fișier */  
            return prim;  
  
        /* stabilește dacă cuvântul curent citit a fost deja întâlnit în text */  
        q = prim; /* căutarea se face începând cu primul cuvânt */  
        while ( q ) { /* dacă q != 0 înseamnă că pointează spre o  
                        data din memoria heap */  
            if(strcmp( q->tp,p) == 0)  
                break; /* cuvântul a mai fost întâlnit în text */  
  
            /* - se trece la data corespunzătoare cuvântului următor citit din textul de  
               intrare;  
               - adresa acestei date este valoarea pointerului urm al datei spre care  
               pointează q;  
               - aceasta se poate exprima cu ajutorul expresiei:  
                   q -> urm;  
               - aceasta adresă se atribuie lui q și în felul acesta se poate relua ciclul */  
  
            q = q -> urm;  
        }  
        if(q) { /* dacă q != 0, cuvântul a mai fost întâlnit în textul  
                de intrare */  
            q -> nc++; /* se incrementează numărul de apariții al  
                        cuvântului */  
            free(p); /* se eliberează zona ocupată de cuvânt */  
        }  
        else { /* cuvântul este nou */
```

```

/* se rezerva zona pentru o data de tip TR */
q = (TR *)malloc(sizeof(TR));
if(q == 0 ) { /* nu se poate aloca memoria heap */
    printf("memorie insuficienta\n");
    exit(1);
}

/* se pastreaza pointerul spre cuvintul nou */
q -> tp = p;

/* nc = 1 */
q -> nc = 1;
if(ultim ) /* daca ultim != 0, cuvintul curent nu este
            primul cuvint citit */
/* ultim pointeaza spre data din memoria heap corespunzatoare
   cuvintului precedent pastrat, deci
   ultim -> urm = q
*/
    ultim -> urm = q;

/* - q -> urm = 0, deoarece data spre care pointeaza q corespunde
   ultimului cuvint citit si pastrat in memoria heap;
   - din aceleasi motive
   ultim = q
*/
    q -> urm = 0;
    ultim = q;
    if(prim == 0 ) /* cuvintul curent este primul cuvint
                    citit */
        prim = q;
    } /* sfirsit else */
} /* sfirsit for */
}

```

10.52 Să se scrie un program care citește și afișează frecvența de apariție a cuvintelor din textul citit, folosind funcția *pfrecvcuv* definită în exercițiul precedent.

În programul de față nu mai este necesar să ne limităm la 100 de cuvinte, cum am făcut în programul din exercițiul 10.50. Datele se păstrează în memoria heap pe măsură ce este nevoie de ele.

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>

typedef struct tr {
    char *tp;
    int nc;
    struct tr *urm;
} TR;

#include "bx48.cpp" /* citcuv */
#include "bx51.cpp" /* pfrecvcuv */

main() /* afiseaza frecventa de aparitie a cuvintelor unui text */
{
    TR *p;
    int i;

    p = pfrecvcuv();
    i = 1;

    while( p ) {
        printf("%s\t%d\n",p -> tp, p -> nc);
        if(i%23 == 0 ) {
            printf("actionati o tasta pentru a\
                continua\n");
            getch();
        }

        i++;
        p = p -> urm;
    }
}

```

#### Observație:

Datele de tip TR, păstrate în memoria *heap*, sînt structuri recursive care formează o mulțime ordonată. Există o *primă* structură și fiecare structură are un element următor, exceptînd ultima. De aceea, între elementele acestei mulțimi de structuri este stabilită o relație de ordine. Relația

respectivă se definește printr-un pointer care este componentă a tipului comun al acestor structuri.

O astfel de mulțime se spune că formează o *listă simplu înlănțuită*.

## 11. LISTE

*Lista* este o mulțime *dinamică*, înțelegînd prin aceasta că ea are un număr variabil de elemente. La început lista este o mulțime vidă. În procesul execuției programului se pot adăuga elemente noi listei și totodată pot fi eliminate diferite elemente din listă de care nu mai este nevoie.

Elementele unei liste sînt date de un același tip. În forma cea mai generală, tipul comun elementelor unei liste este un tip utilizator. Deci, elementele unei liste sînt structuri de un același tip. Acest fapt ne conduce la ideea de a organiza o listă sub formă de tablou. Acest lucru este posibil, dar de obicei nu este eficient, deoarece lista are o natură dinamică, ori tablourile în limbajul C nu sînt dinamice. La compilare este necesar să se definească un maxim pentru numărul elementelor unui astfel de tablou, maxim care uneori nu se poate stabili dinainte. Se poate adopta o soluție de compromis dacă se păstrează elementele listei în memoria *heap*, iar într-o memorie statică sau pe stivă se alocă un tablou de pointeri spre elementele listei. În acest caz, elementele tabloului de pointeri ocupă fiecare 16 biți sau 32 de biți dacă pointerii respectivi sînt de tip *far*. Tabloul de pointeri se va dimensiona la un număr de elemente suficient de mare. Avantajul păstrării elementelor listei în memoria *heap* rezultă din faptul că, supradimensionarea tabloului de pointeri nu conduce la un consum prea mare de memorie deoarece elementele tabloului sînt pointeri și nu însăși elementele listei. În același timp, memoria *heap* este gestionată optim, deoarece ea conține, în fiecare moment al execuției programului, numai elementele necesare din listă. Cu toate acestea, există situații în care este dificil a evalua numărul maxim al elementelor unei liste, precum și cazuri cînd numărul lor diferă mult de la execuție la execuție. De aceea, apare problema de a organiza altfel mulțimile de tip listă. Se obișnuiește să se ordoneze elementele unei liste folosind pointeri care intră în compunerea elementelor listei. Datorită acestor pointeri, elementele listei devin structuri recursive. Listele organizate în acest fel se numesc *liste înlănțuite*.

În concluzie, o mulțime dinamică de structuri recursive de același tip, pentru care sînt definite una sau mai multe relații de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numește *listă înlănțuită*.

Elementele unei liste, de obicei, se numesc *noduri*.

Dacă între nodurile unei liste există o singură relație de ordine, atunci lista se numește *simplu înlănțuită*. În mod analog, lista este *dublu înlănțuită* dacă între nodurile ei sînt definite două relații de ordine.

În general, vom spune că o listă este *n-înlănțuită* dacă între nodurile ei



sînt definite  $n$  relații de ordine.

În legătură cu listele înlănțuite se au în vedere unele operații de interes general:

- a. crearea unei liste înlănțuite;
- b. accesul la un nod oarecare al unei liste înlănțuite;
- c. inserarea unui nod într-o listă înlănțuită;
- d. ștergerea unui nod dintr-o listă înlănțuită;
- e. ștergerea unei liste înlănțuite.

## 11.1. Lista simplu înlănțuită

Așa cum s-a arătat mai sus, între nodurile unei liste simplu înlănțuite este definită o singură relație de ordonare. De obicei această relație este cea de succesor, adică fiecare nod conține un pointer a cărui valoare reprezintă adresa nodului *următor* din listă. În mod analog se poate defini relația de *precedent*. În cele ce urmează ne vom mărgini numai la liste simplu înlănțuite pentru care nodurile satisfac relația de succesor.

Într-o astfel de listă există totdeauna un nod și numai unul care nu mai are următor (succesor), precum și un nod care nu este următorul (succesorul) nici unui alt nod. Aceste noduri formează *capetele* listei simplu înlănțuite.

Pentru a gestiona nodurile unei liste simplu înlănțuite, vom utiliza doi pointeri spre cele două capete. Numim *prim* pointerul spre nodul care nu este următorul nici unui nod al listei și cu *ultim* pointerul spre nodul care nu are următor (succesor) în listă. Acești pointeri vor fi utilizați în toate funcțiile care le vom avea în vedere în prelucrarea listelor simplu înlănțuite. Ei pot fi definiți fie ca variabile globale, fie ca parametri pentru funcțiile de prelucrare a listei. Se alege alternativa cu variabile globale în cazul în care nu se gestionează mai multe liste simplu înlănțuite în același program. Dimpotrivă, pointerii respectivi se vor transfera prin parametri în cazul în care programul gestionează mai multe liste simplu înlănțuite.

În cele ce urmează vom considera cazul în care pointerii *prim* și *ultim* sînt variabile globale.

Tipul unui nod într-o listă simplu înlănțuită se poate defini folosind o declarație de forma:

```
typedef struct tnod {  
    declarații
```

```

    struct tnod *urm;
} TNOD;

```

Pointerii *prim* și *ultim* se declară în afara oricărei funcții prin:

```

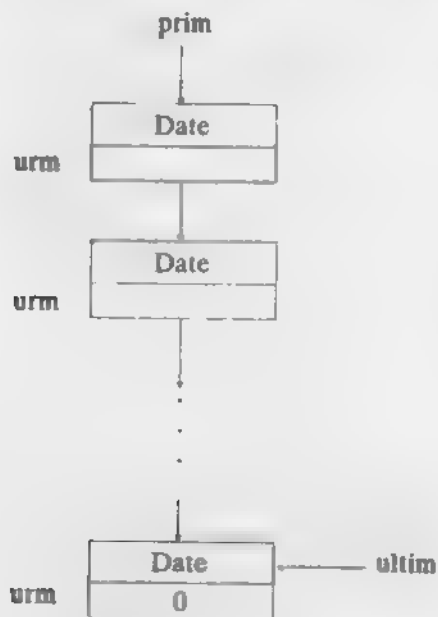
TNOD *prim,*ultim;

```

De obicei, ei vor fi declarați înaintea definirii funcției *main* a programului.

Pointerul *urm* definește relația de succesori pentru nodurile listei. Pentru fiecare nod, el are ca valoare adresa nodului următor din listă. Excepție de la această regulă o constituie nodul spre care pointează variabila *ultim*. În acest caz *urm* are valoarea zero (pointerul nul).

În paragrafele următoare vom construi funcții care să realizeze operațiile a-e, amintite mai sus, pentru listele simplu înlanțuite.



**Listă simplu înlanțuită**

### 11.1.1. Crearea unei liste simplu înlanțuite

La crearea unei liste simplu înlanțuite se realizează următoarele:

1. Se inițializează pointerii *prim* și *ultim* cu valoarea zero, deoarece lista la început este vidă.
2. Se rezervă zonă de memorie în memoria *heap* pentru nodul curent.
3. Se încarcă nodul curent cu datele curente, dacă există și apoi se trece la pasul 4.  
Altfel lista este creată și se revine din funcție.
4. Se atribuie pointerului  
    *ultim* -> *urm*  
adresa din memoria *heap* a nodului curent, dacă lista nu este vidă.  
Altfel se atribuie lui *prim* această adresă.
5. Se atribuie lui *ultim* adresa nodului curent.
6. *ultim* -> *urm* = 0
7. Procesul se reia de la punctul 2 de mai sus pentru a adăuga un nod nou la listă.

Pentru a încărca datele curente în nod (punctul 3) vom apela o funcție care este specifică aplicației. Această funcție poate să aibă un nume dinainte precizat sau să fie transferată printr-un parametru la funcția de creare a listei. Acest parametru va fi un pointer spre funcția respectivă. Soluția cu parametru se alege de obicei când programul prelucrează mai multe liste. În cazul de față am presupus că programul prelucrează o singură listă și de aceea în loc să utilizăm un parametru pointer, vom considera că datele se încarcă prin funcția de nume *incnod*. Ea returnează una din următoarele trei valori: 0, 1 și -1 și anume:

- |    |   |
|----|---|
| 0  | - La eroare.                                      |
| 1  | - La încărcare normală a datelor în nodul curent. |
| -1 | - Când nu mai sînt date de încărcat în nod.       |

De exemplu, dacă funcția *incnod* citește datele pe care le încarcă în nod dintr-un fișier, atunci ea va returna valoarea -1 la întîlnirea sfîrșitului de fișier (nu mai sînt date de încărcat în nod).

Funcția *incnod* are un singur parametru și anume adresa din memoria *heap* rezervată pentru nodul curent, deci acest parametru este un pointer spre tipul comun nodurilor structurii, adică spre *TNOD*.

Din cele de mai sus rezultă că funcția *incnod* are prototipul:

**int incnod(TNOD \*p);**

O altă funcție specifică aplicației concrete și care se apelează la crearea listei este cea care eliberează zona de memorie rezervată nodului curent. Vom numi *elibnod* această funcție. Ea are prototipul:

```
void elibnod(TNOD *p);
```

Această funcție se apelează după un apel al funcției *incnod* în care aceasta nu a încărcat date în nodul curent (s-a revenit din ea cu valoarea 0 sau -1). În acest caz există zonă de memorie rezervată în memoria *heap* pentru nodul curent, dar la apelul funcției *incnod* nu s-au încărcat date ori s-au încărcat date parțial. De aceea, în acest caz vom elibera zona de memorie rezervată apelînd *elibnod*.

Funcția pentru crearea listei simplu înlanțuite o numim *crelist*. Ea returnează una din valorile 0 sau -1 și anume:

- 0 - La eroare.
- 1 - La crearea normală a listei.

Ținînd seama de cele de mai sus, definim funcția *crelist* astfel:

```
int crelist() /*- creaza o lista simplu inlantuita;
               - returneaza:
                 0 - la eroare;
                 -1 - creare normala. */
{
    extern TNOD *prim,*ultim;
    int i,n;
    TNOD *p;

    n=sizeof(TNOD);
    prim=ultim=0; /* initial lista este vida */

    /* se rezerva n octeti pentru nod in memoria heap */
    for(;;){
        if((p=(TNOD *)malloc(n))==0){
            printf("memorie insuficienta la crearea\
                listei\n");
            exit(1);
        }

        /* se incarca date in nod */
        if((i=incnod(p))!=1){
            elibnod(p); /* se elibereaza zona rezervata pentru nod
                        deoarece nu s-au incarcate date in nod */
            return i;
        }
    }
}
```

```

    }

    /* se inlantuie nodul in lista */
    if(ultim!=0)

    /* lista nu este vida */
        ultim -> urm=p;
    else
    /* lista vida */
        prim=p;

    ultim=p;
    ultim -> urm=0;
} /* sfirsit for */
} /* sfirsit crelist */

```

Aşa cum s-a arătat înainte, o listă poate fi organizată păstrind elementele ei în memoria *heap*, iar adresele lor într-o tabelă de pointeri.

Dăm mai jos o variantă a funcţiei *crelist* care să creeze o listă după acest principiu. În acest caz, tabloul de pointeri va fi global şi are un număr maxim de elemente.

Numim *tpnod* acest tablou. De asemenea, vom nota cu *itpnod* variabila globală care are ca valoare indicele primului element al tabloului *tpnod* care este liber. Cu alte cuvinte, *tpnod[itpnod-1]* este pointerul spre ultimul nod adăugat la listă. Iniţial *upnod* are valoarea zero.

Tipul nodurilor nu mai este necesar să fie recursiv, deoarece în acest caz ordonarea se realizează prin indici:

- primul nod are indicele zero;
- al doilea nod are indicele 1;
- ...
- ultimul nod are indicele *itpnod-1*.

De aceea, în locul tipului *TNOD* vom folosi tipul:

```

typedef struct {
    declarati
} TTNOD;

```

Tabloul *tpnod* se declară astfel:

```

TTNOD *tpnod[MAX];

```

unde:

**MAX** - Este o constantă simbolică în prealabil definită printr-o construcție *#define* și valoarea ei se va alege așa încît să nu fie depășită de numărul elementelor listei.

Variabila *itpnod* este de tip *int*:

```
int itpnod;
```

Indicăm mai jos funcția pentru crearea listei cu utilizarea tabloului *itpnod*. Ea utilizează funcțiile *incnod* și *elibnod* ca în cazul funcției *crelist*, fiind analogă cu aceasta.

```
int tpcrelis() /*- creaza o lista folosind un tablou de pointeri
               care contine adresele nodurilor listei;
               - functia returneaza:
                 0 - la eroare;
                 -1 - la creare normala. */
{
    extern TNOD *tpnod[];
    extern int itpnod;
    int i,n;
    TNOD *p;

    n=sizeof(TNOD);
    for(itpnod=0; itpnod<MAX; itpnod++){
        if((p=(TNOD *)malloc(n))==0){
            printf("memorie insuficienta\n");
            exit(1);
        }

        /* se incarca date in nod */
        if((i=incnod(p))!=1){
            elibnod(p);
            return i;
        }
        tpnod[itpnod]=p; /* pastreaza adresa nodului in tabloul de
                           pointeri */
    } /* sfirsit for */
}
```

### 11.1.2. Accesul la un nod al unei liste simplu înlanțuite

Putem avea acces la nodurile unei liste simplu înlanțuite începînd cu nodul spre care pointează variabila globală *prim* și trecînd apoi pe rînd de la un nod la altul, folosînd pointerul *urm*.

Există cazuri în care dorim acces la un nod anumit al listei. În acest caz, este necesar să definim modul de indentificare al nodului la care dorim să avem acces. Un mod simplu este acela de a indica numărul de ordine al nodului la care se dorește acces, de exemplu, se dorește acces la al  $n$ -lea nod al listei. Cum lista este o mulțime dinamică, acest mod de a defini accesul la un nod al ei nu este totdeauna cel mai nimerit. O altă metodă este aceea, de a avea o dată componentă a nodurilor, care să aibă valori diferite, pentru noduri diferite. În acest caz se poate defini accesul la nodul din listă pentru care data respectivă are o valoare dată. O astfel de dată, care este componentă a nodurilor unei liste și are valori distincte pentru noduri diferite ale unei liste se numește *cheie*.

Cheia poate fi o dată de un tip oarecare.

În cazul de față vom considera cheia de tip *int*. Cititorul poate modifica simplu funcția de mai jos pentru a utiliza chei de alte tipuri: *long*, *char*, *double* etc.

Funcția returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conține un nod a cărui cheie să aibă valoarea indicată de parametrul ei.

Nodurile listei au tipul definit astfel:

```
typedef struct tnod{
    declarati
    int cheie;
    declarati
    struct tnod *urm;
}TNOD;
```

Putem acum defini funcția de căutare ca mai jos:

```
TNOD *enci(int c) /* - cauta un nod al listei pentru care cheie=c;
                    - returneaza pointerul spre nodul determi-
                      nat in acest fel sau zero daca nu exista nici
                      un nod asa incit cheia sa aiba valoarea c. */
{
    extern TNOD *prim;
    TNOD *p;

    p=prim; /* cautarea incepe cu nodul spre care pointeaza variabila
              prim */
    while(p!=0){
        if(p->cheie==c)
            return p; /* s-a gasit un nod pentru care cheie=c */
    }
```

```

        p=p ->urm; /* se trece la nodul urmator din lista */
    }
    /* in acest punct se ajunge cind nu exista un nod in lista cu proprietatea
       ca cheie=c */
    return 0;
}

```

O funcție similară se poate construi și pentru cazul în care la implementarea listei se utilizează tabloul de pointeri *tpnod* (vezi paragraful precedent).

În acest caz tipul nodurilor listei se declară astfel:

```

typedef struct {
    declarații
    int cheie;
    declarații
} TTNOD;

```

Funcția caută nodul din listă parcurgînd-o cu ajutorul indicelui care variază de la zero pînă la *itpnod-1* inclusiv.

```

TTNOD *tpnci(int c)
/*- cauta un nod al listei pentru care cheie=c;
   - returneaza pointerul spre nodul respectiv sau zero daca nu exista in
     lista un astfel de nod */
{
    extern TTNOD *tpnod[];
    extern int itpnod;
    int i;

    for(i=0;i<itpnod;i++)
        if(tpnod[i] -> cheie==c)
            return tpnod[i];
    return 0;
}

```

Ambele funcții parcurg elementele listei, nod cu nod, fie pînă la întîlnirea nodului căutat, fie pînă la sfîrșitul listei dacă nu există în listă un nod a cărui cheie să aibă valoarea cerută. Această metodă de căutare se numește *căutare liniară*. Ea este o metodă foarte simplă de căutare dar nu este eficientă și de aceea se aplică numai la liste cu un număr relativ mic de noduri.

O metodă mult mai eficientă este metoda *căutării binare*. Această metodă poate fi aplicată simplu la listele implementate cu ajutorul tabloului de pointeri *tpnod*. În acest scop lista trebuie întîi sortată. de exemplu



crescător, în raport cu valorile cheii. Sortarea se poate realiza procedind ca în exercițiul 8.18. în care cuvintele citite de la intrarea standard se păstrează în memoria *heap* și în paralel cu aceasta se definește un tablou cu adresele la care sînt păstrate aceste cuvinte. Apoi se realizează sortarea lor (ordonarea în ordine alfabetică) folosind metoda bulelor.

Pentru o eficiență mai bună se poate utiliza o altă metodă de sortare (sortare *shell*, sortare rapidă, sortare cu ajutorul arborilor etc.).

### 11.1.3. Inserarea unui nod într-o listă simplu înlănțuită

Într-o listă simplu înlănțuită se pot face inserări de noduri în diferite poziții. Amintim cîteva posibilități care intervin mai frecvent:

- a. inserare înaintea primului nod;
- b. inserare înaintea unui nod precizat printr-o cheie;
- c. inserare după un nod precizat printr-o cheie;
- d. inserare după ultimul nod al listei (adăugarea unui nod nou la listă).

În cele ce urmează se definesc funcții pentru cazurile a-d indicate mai sus, atît pentru liste simplu înlănțuite, cit și variantele lor cînd lista se implementează cu ajutorul tabloului de pointeri *tpnod*.

Tipurile TNOD și TTNOD se consideră declarate ca în paragrafele precedente.

#### 11.1.3.1. Inserarea unui nod într-o listă simplu înlănțuită înaintea primului ei nod

Primul nod al unei liste simplu înlănțuite este nodul spre care nu pointează pointerul *urm* al nici unui nod al listei, adică este nodul care nu este succesorul nici unui nod din listă. Amintim că spre acest nod pointează variabila *prim* dacă lista nu este vidă.

În cazul utilizării tabloului *tpnod*, primul nod al listei este nodul spre care pointează elementul:

`tpnod[0]`

Funcția de inserare returnează pointerul spre nodul inserat în listă sau zero în cazul în care nu se poate realiza inserarea. De altfel, toate funcțiile de inserare returnează aceste valori.

```
TNOD *iniprim() /* insereaza nodul curent inaintea primului
                  nod al listei */
{
```

```

extern TNOD *prim,*ultim;
TNOD *p;
int n;
n=sizeof(TNOD);

/* rezerva memorie heap si incarca datele in zona respectiva */
if((p=(TNOD *)malloc(n))!=0)&&(inchnod(p)==1)){
    if(prim==0){ /* lista vida */

/* lista se compune numai din nodul curent */
        prim=ultim=p;
        p -> urm=0; /* nu exista succesor */
    }
    else{
        p -> urm=prim; /* primul nod al listei devine succesorul
                        nodului care se insereaza */
        prim=p; /* nodul inserat nu este succesorul nici unui alt
                nod si de aceea prim pointeaza spre el */
    }
    return p;
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
/*s-a rezervat memorie pentru nod dar nu s-au incarat date in nod */
elibnod(p);
return 0;
}

```

Inserarea în cazul utilizării tabloului *tpnod* implică eliberarea elementului *tpnod[0]*. În acest scop se fac atribuiri:

*tpnod[i]=tpnod[i-1]* pentru *i=itpnod, itpnod-1, ..., 2, 1*

Se observă că funcția de inserare înaintea primului nod pentru liste, care folosește tabloul de pointeri *tpnod* nu mai este așa de eficientă ca funcția *iniprim* definită mai sus.

```

TNOD *tpiniprim() /* insereaza un nod inaintea primului
                  nod al listei */
{
    extern TNOD *tpnod[];
    extern int itpnod;
    int n;

```

```

int i;

if(itpnod >= MAX){
    printf("lista are prea multe elemente\n");
    return 0;
}
n=sizeof(TTNOD);
if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1))
{
    /* deplaseaza valorile elementelor tabloului tpnod */
    for(i=itpnod++;i>0;i--)
        tpnod[i]=tpnod[i-1];
    tpnod[0]=p;
    return p;
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

### 11.1.3.2. Inserarea unui nod într-o listă simplu înălănțuită înaintea unui nod precizat printr-o cheie

Vom presupune că,cheia este de tip *int* ca și în cazul funcției *enci* definită în paragraful 11.1.2.

```

TNOD *inici(int c)
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numerică;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea
   nu are loc. */
{
    extern TNOD *prim;
    TNOD *q,*q1,*p;
    int n;

    n=sizeof(TNOD);
    /* - cauta nodul de cheie=c;
       - la terminarea ciclului:
           q - pointeaza spre nodul respectiv;
           q1 - pointeaza spre nodul precedent celui spre care pointeaza q. */
    q1=0;

```

```

q=prim;
while(q){
    if(q -> cheie==c)
        break; /* s-a gasit nodul cautat */
    q1=q;
    q=q -> urm;
}
if(q==0){ /* nu exista in lista un nod de cheie=c */
    printf("nu exista in lista un nod de\
        cheie=%d\n",c);
    return 0;
}

/* se rezerva zona pentru nod si se incarca datele nodului in zona
respectiva */
if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
    if(q==prim){

/*cheie=c pentru primul nod si inserarea se face inaintea
primului nod */
        p -> urm=prim;
        prim=p;
    }
    else{

/*nodul spre care pointeaza p se insereaza intre nodul spre care
pointeaza q1 si nodul spre care pointeaza q */
        q1-> urm=p; /* succesorul nodului spre care pointeaza
            q1 este nodul spre care pointeaza p */
        p -> urm=q; /* succesorul nodului spre care pointeaza p
            este nodul spre care pointeaza q */
    }
    return p;
}

/* nu s-a facut inserarea ceruta */
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

În continuare definim funcția care utilizează tabloul de pointeri *tpnod*.

```
TTNOD *tpinici(int c)
```

```
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numerica;  
   - returneaza pointerul spre nodul inserat sau zero daca nu are  
   loc inserarea. */
```

```
{  
    extern TTNOD *tpnod;  
    extern int itpnod;  
    int i,k;  
  
    /*daca itpnod >= MAX nu exista loc liber in tabloul de pointeri */  
    if(itpnod >= MAX)  
        printf("lista are prea multe elemente\n");  
        return 0;  
}
```

```
/* cauta nodul de cheie=c */  
for(i=0;i<itpnod;i++)  
    if(tpnod[i] -> cheie==c)  
        break; /* s-a gasit nodul cautat */  
if(i==itpnod){ /* nu exista in lista un nod de cheie=c */  
    printf("nu exista in lista un nod de\  
           cheie=%d\n",c);  
    return 0;  
}
```

```
n=sizeof(TTNOD);
```

```
/* rezerva zona pentru nod si se incarca datele nodului in zona  
   respectiva */
```

```
if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1)){
```

```
/* deplaseaza valorile elementelor tabloului tpnod:
```

```
    tpnod[k]=tpnod[k-1], pentru k=itpnod,itpnod-1,...,i+1. */  
    for(k=itpnod++;k>i;k--)  
        tpnod[k]=tpnod[k-1];
```

```
/* tpnod[i] devine egal cu adresa nodului care se insereaza */
```

```
    tpnod[i]=p;  
    return p;
```

```
}
```

```
/* nu s-a facut inserarea */
```

```
if(p==0){
```

```

        printf("memorie insuficienta\n");
        exit(1)
    }
    elibnod(p);
    return 0;
}

```

### 11.1.3.3. Inserarea unui nod într-o listă simplu înlanțuită după un nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*.

```

TNOD *indci(int c)
/* - insereaza un nod dupa un nod precizat printr-o cheie numerica;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu
   are loc. */
{
    extern TNOD *prim,*ultim;
    TNOD *p,*q;
    int n;

    /* cauta nodul de cheie=c */
    for(q=prim;q=q->urm)
        if(q->cheie==c)
            break;
    if(q==0){
        printf("nu exista in lista un nod de\
              cheie=%d\n",c);
        return 0;
    }

    /* se rezerva zona pentru nod si se incarca datele in nod */
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){

        /* se insereaza nodul spre care pointeaza p dupa nodul spre care
        pointeaza q */
        p->urm=q->urm; /* nodul succesor nodului spre care
                        pointeaza q devine succesor nodului
                        spre care pointeaza p */
        q->urm=p; /* succesorul nodului spre care pointeaza q
                  devine nodul spre care pointeaza p */
        if(ultim==q) /* - nodul spre care pointeaza q nu a avut suc-

```

```

        cesor;
        - in acest moment nodul spre care pointeaza p nu are succesor. */
        ultim=p;
        return p;
    }

    /* nu s-a reusit inserarea nodului */
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

Funcția de inserare pentru liste implementate cu ajutorul tabloului de pointeri *tpnod* care realizează inserarea după un nod de cheie dată este asemănătoare cu funcția *tpnuci* definită în paragraful precedent. În acest caz, se modifică instrucțiunea *for* pentru deplasarea elementelor tabloului *tpnod* și instrucțiunea de atribuire următoare ei:

```

for(k=itpnod++;k>i+1;k--)
    tpnod[k]=tpnod[k-1];
tpnod[i+1]=p;

```

#### 11.1.3.4. Adăugarea unui nod la o listă simplu înlănțuită

Adăugarea unui nod la o lista simplu înlănțuită înseamnă inserarea lui după nodul spre care pointeaza variabila *ultim*. Aceasta înseamnă că după inserarea nodului, variabila *ultim* pointează spre nodul respectiv, acesta devenind nodul din listă care nu are succesor.

```

TNOD *adauga( )
/*- adauga un nod la o lista simplu inlantuita;
- returneaza pointerul spre nodul inserat sau zero daca nu se realizeaza
  inserarea. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    /* se rezerva zona de memorie pentru nod si se incarca datele in zona
       respectiva */
    n=sizeof(TNOD);
    if((p=(TNOD *)malloc(n))!=0) { if (incnod(p)==1){

```

```

    if (prim==0) /* lista este vida */
        prim=ultim=p;
    else{
        ultim -> urm=p; /* succesul nodului spre care
                           pointeaza ultim devine nodul
                           spre care pointeaza p */
        ultim=p; /* acesta devine nodul spre care
                   pointeaza ultim */
    }
    p -> urm=0; /* nodul spre care pointeaza p nu are succesul */
    return p;
}

/* nu s-a reusit inserarea nodului in lista */
if (p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

Definim mai jos funcția de adăugare a unui nod la o listă implementată cu ajutorul tabloului *tpnod*.

```

TTNOD *tpadauga()
/*- adauga un nod la o lista;
  - returneaza pointerul la nodul inserat sau zero daca inserarea
  nu se realizeaza. */
{
    extern TTNOD *tpnod[];
    extern int itpnod;
    int n;

    if (itpnod >= MAX){
        printf("lista are prea multe elemente\n");
        return 0;
    }
    n=sizeof(TTNOD);
    if ((p=TTNOD *)malloc(n))!=0 && (incnod(p)==1){
        tpnod[itpnod++]=p;
        return p;
    }
    if (p==0){

```



```

        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod (p);
    return 0;
}

```

### 11.1.4. Ștergerea unui nod dintr-o listă simplu înlănțuită

Dintr-o listă simplu înlănțuită se pot șterge noduri. Aceasta se poate realiza în mai multe moduri. În cele ce urmează avem în vedere următoarele cazuri:

- ștergerea primului nod al listei simplu înlănțuite;
- ștergerea unui nod precizat printr-o cheie;
- ștergerea ultimului nod al listei simplu înlănțuite.

Funcțiile de ștergere utilizează funcția *elibnod*. Această funcție eliberează zona de memorie alocată nodului care se șterge, precum și eventualele zone de memorie alocate suplimentar prin intermediul funcției *incnod* pentru a păstra diferite componente ale unui nod (de exemplu componente de tip șir de caractere).

Alături de funcțiile obișnuite de ștergere, se definesc astfel de funcții și pentru listele implementate cu ajutorul tabloului *tpnod*.

#### 11.1.4.1. Ștergerea primului nod al unei liste simplu înlănțuite

```

void apn() /* șterge primul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if(prim==0) /* lista vida */
        return;
    p=prim;
    prim=prim -> urm;
    elibnod(p);
    if(prim==0) /* lista a devenit vida */
        ultim=0;
}

```

În cazul în care lista este implementată cu ajutorul tabloului *tpnod*, se elimină nodul spre care pointează elementul *tpnod[0]*, apoi se deplasează

elementele tabloului *tpnod* folosind atribuirile:

```
tpnod[i]=tpnod[i+1], pentru i=0, 1, ..., itpnod-2.  
void tpspn() /* sterge primul nod din lista */  
{  
    extern TNOD *tpnod[];  
    extern int itpnod;  
    int i;  
  
    elibnod(tpnod[0]);  
    for(i=0; i<itpnod-1, i++)  
        tpnod[i]=tpnod[i+1];  
    itpnod--;  
}
```

#### 11.1.4.2. Șterge un nod precizat printr-o cheie, dintr-o listă simplu înălțată

Vom considera că, cheia este de tip *int*. În acest caz tipul TNOD se definește ca în paragraful 11.1.2.

```
voidanci(int c) /* sterge nodul pentru care cheie=c */  
{  
    extern TNOD *prim, *ultim;  
    TNOD *q, *q1;  
  
    q1=0;  
    q=prim;  
  
    /*- se cauta nodul de cheie=c;  
    - la terminarea ciclului q pointeaza spre nodul respectiv sau  
      q=0 daca nu exista un astfel de nod;  
    - q1 pointeaza spre nodul a carui succesor este nodul spre care  
      pointeaza q sau q1=0 daca q=prim. */  
    while(q){  
        if(q->cheie==c)  
            break; /* s-a gasit nodul cautat */  
        q1=q;  
        q=q->urm;  
    }  
    if(q==0){ /* nu exista in lista un nod pentru care cheie=c */  
        printf("lista nu contine un nod de\  
            cheie=%d\n", c);  
        return;  
    }
```

```

}

/* se sterge nodul spre care pointeaza q */
if(q==prim){ /* se sterge primul nod din lista */
    prim=prim -> urm;
    elibnod(q);
    if(prim==0)
        ultim=0; /* lista a devenit vida */
}
else{
    q1->urm=q->urm; /*succesorul nodului spre care pointeaza
                    q1 devine succesorul nodului spre care
                    pointeaza q */
    if(q==ultim) /* se sterge ultimul nod, deci nodul spre care
                  pointeaza q1 devine ultimul nod al listei */
        ultim=q1;
    elibnod(q);
}
}

```

În cazul în care lista este implementată cu ajutorul tabloului *tpnod*, se caută nodul pentru care cheie=*c*. Fie acesta nodul a cărui adresă este dată de elementul *tpnod[i]*.

După eliminarea nodului respectiv, se realizează atribuiri:

*tpnod[k]=tpnod[k+1]*, pentru *k=i, i+1, ..., itpnod-2*.

*void tpanci(int c) /\* sterge nodul de cheie=c \*/*

```

{
    extern TTNOD *tpnod[];
    extern int itpnod;
    int i;

    for(i=0; i<itpnod; i++)
        if(tpnod[i] -> cheie==c)
            break;
    if(i==itpnod){
        printf("nu exista un nod de cheie=%d\n",c);
        return;
    }
    elibnod(tpnod[i]);
    for(; i<itpnod-1; i++)
        tpnod[i]=tpnod[i+1];
    itpnod--;
}

```

```
}
```

#### 11.1.4.3. Ștergerea ultimului nod dintr-o listă simplu înlanțuită

```
void sun() /* sterge ultimul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *q,*q1;

    q1=0;
    q=prim;
    if(q==0)
        return; /* lista vida */
    while(q!=ultim) { /* se parcurge lista pina se ajunge la ultimul
                        nod al ei */
        q1=q;
        q=q -> urm;
    }
    if(q==prim) /* - lista contine un singur nod care se sterge;
                  - lista devine vida. */
        prim=ultim=0;
    else{
        /* - nodul spre care pointeaza q1 are ca succesori nodul spre care
           pointeaza q si acesta este ultimul nod al listei;
           - cum nodul spre care pointeaza q se sterge, nodul spre care pointeaza
           q1 devine ultimul, deci q1 -> urm=0 si ultim=q1. */
        q1 -> urm=0;
        ultim=q1;
    }
    elibnod(q);
}
```

În cazul în care lista se implementează folosind tabloul *tpnod*, operația de ștergere a ultimului nod este mult mai eficientă, deoarece nu necesită parcurgerea nodurilor listei.

```
void tpsun() /* sterge ultimul nod al listei */
{
    extern TTNOD *tpnod[];
    extern int itpnod;

    if(itpnod==0)
        return; /* lista vida */
}
```

```

    elibnod(--itpnod);
}

```

### Observație:

Funcțiile definite în paragrafele precedente realizează operațiile cele mai frecvent utilizate asupra listelor. Unele dintre aceste funcții necesită parcurgerea nodurilor listei, parțial sau în totalitate. Alte funcții nu necesită astfel de parcurgeri. Funcțiile care parcurg nodurile unei liste au o eficiență scăzută în comparație cu cele care nu fac astfel de parcurgeri. De aceea, se recomandă pe cât posibil utilizarea funcțiilor care nu necesită parcurgerea nodurilor ei. Astfel de funcții sunt:

- iniprim* - Inserarea a nodul curent înaintea primului nod al listei (11.1.3.1.).
- adauga* - Adaugă un nod la o listă (11.1.3.4.).
- spn* - Ștergerea primului nod al listei (11.1.4.1).

În cazul listelor implementate cu ajutorul tabloului *tpnod*, cele mai eficiente funcții sunt cele care nu necesită instrucțiuni ciclice care să se execute asupra elementelor tabloului *tpnod*. Astfel de funcții sunt:

- tpadauga* - Adaugă un nod la o listă
- tpsun* - Șterge ultimul nod al listei.

De aceste observații se ține, uneori, seama la alegerea modului de implementare a listelor în aplicații.

## 11.1.5. Ștergerea unei liste simplu înlănțuite

Se utilizează funcția *elibnod* pentru fiecare nod al listei.

```

void streglist() /* sterge o lista simplu inlantuita */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    while(prim){
        p=prim;
        prim=prim -> urm;
        elibnod(p);
    }
    ultim=0;
}

```

Mai jos definim funcția de ștergere a listei implementate cu ajutorul tabloului *tpnod*.

```
void tpsterglist() /* șterge lista implementată cu tablou
                    de pointeri */
{
    extern TNOD *tpnod[];
    extern int itpnod;
    int i;

    for(i=0; i<itpnod; i++)
        elibnod(tpnod[i]);
    itpnod=0;
}
```

### Exerciții:

11.1 Se consideră tipul utilizator:

```
typedef struct tnod{
    char *cuvant;
    int frecventa;
    struct tnod *urm;
}TNOD
```

Se cere să se scrie funcția *incnod* care încarcă datele curente într-un nod de tip TNOD.

Prin cuvânt se înțelege un șir de litere mici sau mari, ca în exercițiul 10.48. În acest exercițiu se definește funcția *cucuv* care citește un cuvânt de la intrarea standard și-l păstrează în memoria *heap*. Funcția respectivă returnează adresa de început a zonei în care se păstrează cuvântul citit sau zero în caz că se întâlnește sfârșitul de fișier.

Funcția de față apelează funcția *cucuv* și atribuie adresa returnată de ea pointerului *cuvant* din nodul curent. De asemenea, se atribuie valoarea 1, variabilei *frecventa*.

Funcția *incnod* returnează valoarea -1 dacă *cucuv* returnează valoarea zero și 1 altfel.

### FUNCȚIA BX11

```
int incnod(TNOD *p)
/* încarcă datele curente în nodul spre care pointează p */
{
```

```

    if((p -> cuvnt = citcuv() ) == 0 )
        return -1;
    p -> frecventa = 1;
    return 1;
}

```

- 11.2 Să se scrie funcția *elibnod* care eliberează zonele din memoria *heap* ocupate de nodul de tip *TNOD* definit în exercițiul precedent.

## FUNCȚIA BX12

```

void elibnod(TNOD *p)
/* elibereaza zonele din memoria heap ocupate de nodul spre
   care pointeaza p */
{
    free( p -> cuvnt);
    free(p);
}

```

- 11.3 Să se scrie funcția *adauga*, care permite adăugarea unui nod de tip *TNOD* la o listă simplu înlănțuită, după ultimul nod al listei. Tipul *TNOD* este cel definit în exercițiul 11.1.

Să observăm că funcția *adauga* definită în paragraful 11.1.3.4. este dependentă numai de pointerul *urm* din tipul nodului. De aceea, funcția *adauga* de mai jos este identică cu cea din paragraful amintit mai sus.

Funcția de față apelează funcțiile *incnod* și *elibnod* definite în exercițiile precedente.

## FUNCȚIA BX13

```

TNOD *adauga()
/* - adauga un nod la o lista simplu inlantuita;
   - returneaza pointerul spre nodul adaugat sau zero daca nu s-a
     realizat adaugarea. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n = sizeof(TNOD);
    if(((p = (TNOD *)malloc(n)) != 0 ) &&
        (incnod(p) == 1 )) {

```

```

    if (prim == 0)
        prim = ultim -> p;
    else {
        ultim -> urm = p;
        ultim = p;
    }
    p -> urm = 0;
    return p;
}

if (p == 0) {
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

11.4 Fie o listă simplu înlanțuită ale cărei noduri au tipul TNOD definit în exercițiul 11.1.

Să se scrie o funcție care caută în lista respectivă, nodul pentru care pointerul *cuvant* are ca valoare adresa unui cuvânt dat.

Cu alte cuvinte, pointerul *cuvant* joacă rol de cheie și se cere să se găsească nodul a cărui cheie pointează spre un cuvânt dat.

Această funcție este asemănătoare cu funcția *enci* definită în paragraful 11.1.2.

## FUNCȚIA BX14

```

TNOD *cncs(char *c)
/* - cauta un nod al listei pentru care cuvintul spre care pointeaza cuvant
   este identic cu cel spre care pointeaza c;
   - returneaza pointerul spre nodul determinat sau zero daca nu exista
   un astfel de nod. */
{
    extern TNOD *prim;
    TNOD *p;

    for (p = prim; p ; p = p->urm)
        if (strcmp(p->cuvant, c) == 0)
            return p;
    return 0;
}

```



11.5 Să se scrie o funcție care șterge ultimul nod al unei liste simplu înlanțuite ale cărei noduri au tipul TNOD definit în exercițiul 11.1.

Această funcție este definită în paragraful 11.1.4.3 și ea este dependentă numai de pointerul *urm* din tipul nodurilor.

Ea apelează funcția *elibnod* definită în exercițiul 11.2.

## FUNCȚIA BXIS

```
void sun() /* șterge ultimul nod din lista */
{
    extern TNOD *prim, *ultim;
    TNOD *q1, *q;

    q1 = 0;
    if(prim == 0 )
        return;
    for(q = prim; q && q != ultim; q=q->urm)
        q1 = q;
    if(q == prim )
        prim = ultim = 0;
    else {
        q1 -> urm = 0;
        ultim = q1;
    }
    elibnod(q);
}
```

11.6 Să se scrie un program care citește cuvintele dintr-un text și afișează numărul de apariții al fiecărui cuvint din textul respectiv.

Cuvîntul se definește ca o succesiune de litere mici și/sau mari. Textul se termină prin sfîrșitul de fișier.

Această problemă a fost rezolvată în exercițiul 10.50. Problema s-a rezolvat utilizînd un tablou de pointeri ale cărui elemente sînt adrese pentru date păstrate în memoria *heap*. Acest tablou este similar cu tabloul *tpnod* introdus în acest capitol și utilizat la implementarea listelor.

În exercițiul de față programul se realizează utilizînd o listă simplu înlanțuită ale cărei noduri au tipul TNOD definit în exercițiul 11.1.

Deși programul din exercițiul 10.50 este mai eficient din punct de vedere al timpului de execuție decît cel de față, el are o limitare cu privire la numărul de cuvinte diferite din text.

În exercițiul 10.50 s-a limitat numărul cuvintelor diferite la 100. Evident, această limită poate fi schimbată simplu, modificând valoarea lui MAX.

În programul de față nu mai este necesar să definim acest maxim. El se execută astfel:

1. La întâlnirea unui cuvânt se construiește un nod pentru cuvântul respectiv.  
Acesta conține pointerul spre cuvânt, care este păstrat în memoria *heap*; frecvența de apariție a cuvântului se face egală cu 1.
2. Nodul construit la punctul 1 se adaugă la lista simplu înlanțuită care se construiește.
3. Se caută în listă un nod care să corespundă cuvântului curent.

Dacă există un astfel de nod și acesta nu este ultimul nod al listei, atunci se mărește frecvența din nodul respectiv și apoi se șterge ultimul nod al listei (cel adăugat la punctul 2) deoarece cuvântul există deja în listă.

După pasul 3 se revine la pasul 1 și ciclul continuă până când nu mai sînt cuvinte de citit (s-a ajuns la sfîrșitul de fișier). În acest moment se listează cuvintele și frecvența lor de apariție, corespunzătoare nodurilor listei.

## PROGRAMUL BXI6

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bxi1.cpp" /* incnod */
#include "bxi2.cpp" /* elibnod */
#include "bxi3.cpp" /* adauga */
#include "bxi4.cpp" /* cncs */
#include "bxi5.cpp" /* sun */

TNOD *prim,*ultim;
```

```

main() /* citește un text și afișează frecvența cuvintelor din text */
{
    TNOD *p,*q;

    prim = ultim = 0; /* la început lista este vidă */
    while((p = adauga()) != 0 )

    /* s-a adăugat la lista un nod corespunzător ultimului cuvânt citit */
        if((q=cncs(p->cuvant)) != ultim) {

    /* - cuvântul există într-un nod care nu este ultimul nod al listei;
       - deci există deja în listă;
       - se mărește frecvența lui și se șterge ultimul nod al listei. */
            q -> frecventa++;
            sun();
        }

    /* listează cuvintele și frecvența lor */
    for(p = prim; p ; p = p->urm)
        printf("cuvântul: %-51s are frecvența: %d\n",
            p -> cuvant, p -> frecventa);
}

```

- 11.7 Să se scrie un program care citește cuvintele dintr-un text și scrie numărul de apariții al fiecărui cuvânt în ordinea alfabetică a cuvintelor respective.

Acest program este asemănător cu cel precedent. Diferența constă în faptul că înainte de a lista cuvintele și frecvența lor, se face o ordonare a nodurilor listei în așa fel încât cuvintele corespunzătoare nodurilor să fie ordonate alfabetic. Aceasta se realizează cu ajutorul funcției *ordlist* definită mai jos.

Ordonarea nodurilor se face pe baza schimbării înălțurii acestora.

Se utilizează metoda de sortare a bulelor. Potrivit acestei metode, se compară cuvintele corespunzătoare a două noduri vecine. Dacă ele nu sînt în ordine alfabetică, atunci se schimbă ordinea de înălțuire în listă a nodurilor respective.

## PROGRAMUL BX17

```

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>

```

```

#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bx11.cpp" /* incnod */
#include "bx12.cpp" /* elibnod */
#include "bx13.cpp" /* adauga */
#include "bx14.cpp" /* cncs */
#include "bx15.cpp" /* sun */

void ordlist()
/* ordoneaza nodurile listei de tip TNOD in ordinea alfabetica a
   cuvintelor corespunzatoare */
{
    extern TNOD *prim, *ultim;
    TNOD *p, *p1, *q;
    int ind;

    if(prim == 0 )
        return; /* lista vida */
    ind = 1;
    while ( ind ) { /* 1 */

/* cit timp ind nu este zero, trebuie sa se parcurga lista si sa se inverseze
   inlantuirile nodurilor vecine carora le corespund cuvinte care nu sint
   in ordine alfabetica */
        ind = 0;
        p = prim; /* p pointeaza spre nodul curent */
        p1 = 0; /* p1 pointeaza spre nodul precedent */
        q = p -> urm; /* q pointeaza spre nodul urmator */
        while( q != 0 ) /* exista nod urmator */
            if(strcmp(p->cuvant, q->cuvant) > 0) {

/* - cuvintele corespunzatoare nodului curent si nodului urmator nu sint
   in ordine alfabetica;
   - se inverseaza inlantuirile lor. */
                if(p == prim) /* nodul curent nu are precedent */

```

```

        prim = q; /* q devine primul nod al listei */
    else
        p1 -> urm = q; /* q devine urmatorul lui p1 */
    p -> urm = q -> urm; /* urmatorul lui p devine
                           urmatorul lui q */
    q -> urm = p; /* urmatorul lui q devine p */
    p1 = q; /* precedentul lui p devine q */
    q = p -> urm; /* q devine urmatorul lui p */
    if( q == 0)
        ultim = p; /* p nu mai are urmator */
    ind = 1; /* s-a facut o permutare, deci procesul de
              ordonare nu este terminat */
} /* sfirsit if */
else { /* - nu se schimba inlantuirile nodurilor deoarece
        cuvintele corespunzatoare sint in ordine alfa-
        betica;
        - se avanseaza in lista la perechea urmatoare
        de noduri. */
    p1 = p;
    p = q;
    q = q -> urm;
} /* sfirsit else */
} /* sfirsit while 1 */
} /* sfirsit ordlist */

TNOD *prim,*ultim;

main() /* citeste un text si afiseaza frecventa cuvintelor din textul
        respectiv in ordine alfabetica */
{
    TNOD *p;

    prim = ultim = 0;

    /* creeaza lista */
    while( adauga() != 0 )
        if((p = cnsc(ultim -> cuvant)) != ultim ) {
            p -> frecventa++;
            sun();
        }
}

```

```

/* sortare */
ordlist();

/* listare */
for( p=prim; p; p = p->urm)
    printf("cuvintul: %-5ls are frecventa: %d\n",
        p -> cuvnt, p -> frecventa);
}

```

## 11.2. Stive și cozi

În exercițiul 7.1 s-a implementat o stivă cu ajutorul unui tablou. În general, o stivă se implementează printr-o listă simplu înlănțuită.

O *stivă* este o listă simplu înlănțuită gestionată conform principiului LIFO (Last in First Out).

Conform acestui principiu, ultimul nod pus în stivă este primul care este scos din stivă.

Stiva, ca și lista are două capete, *baza stivei* și *virful stivei*.

Asupra unei stive se definesc câteva operații, dintre care cele mai importante sînt:

1. pune un element pe stivă (*push*);
2. scoate un element din stivă (*pop*);
3. șterge (videază) stiva (*clear*).

Primele două operații se realizează în virful stivei. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din virful stivei și în continuare, cel pus anterior lui pe stivă ajunge în virful stivei.

Dacă un element se pune pe stivă, atunci acesta se pune în virful stivei.

Pentru a implementa o stivă printr-o listă simplu înlănțuită va trebui să identificăm baza și virful stivei cu capetele listei simplu înlănțuite. Există două posibilități:

- a nodul spre care pointează variabila *prim* este baza stivei, iar nodul spre care pointează variabila *ultim* este virful stivei;
- b nodul spre care pointează variabila *prim* este virful stivei, iar nodul spre care pointează variabila *ultim* este baza stivei.

În cazul *a*, funcțiile *push* și *pop* se indentifică prin funcțiile *adauga* și respectiv *sun*, definite în paragrafele 11.1.3.4 și respectiv 11.1.4.3.

Dacă funcția *adauga* este eficientă, în schimb funcția *sun* nu este eficientă.

În cazul *b*, funcțiile *push* și *pop* se indentifică prin funcțiile *iniprim* și respectiv *spn*, definite în paragrafele 11.1.3.1. și respectiv 11.1.4.1.

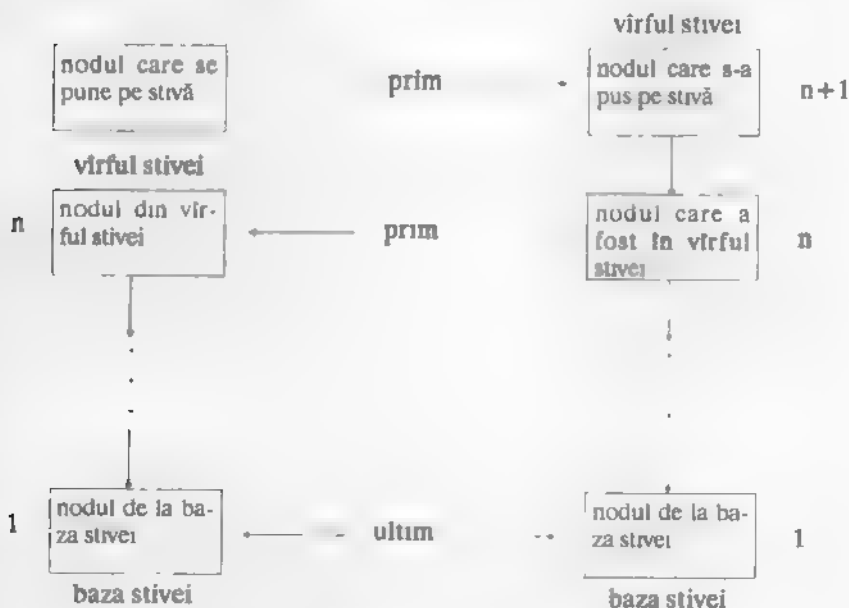
În acest caz ambele funcții sînt eficiente. De aceea, se recomandă implementarea stivei printr-o listă simplu înlănțuită conform cazului *b* indicat mai sus.

Cele două operațiuni *push* și *pop* pot fi schematizate ca în figura de mai jos.

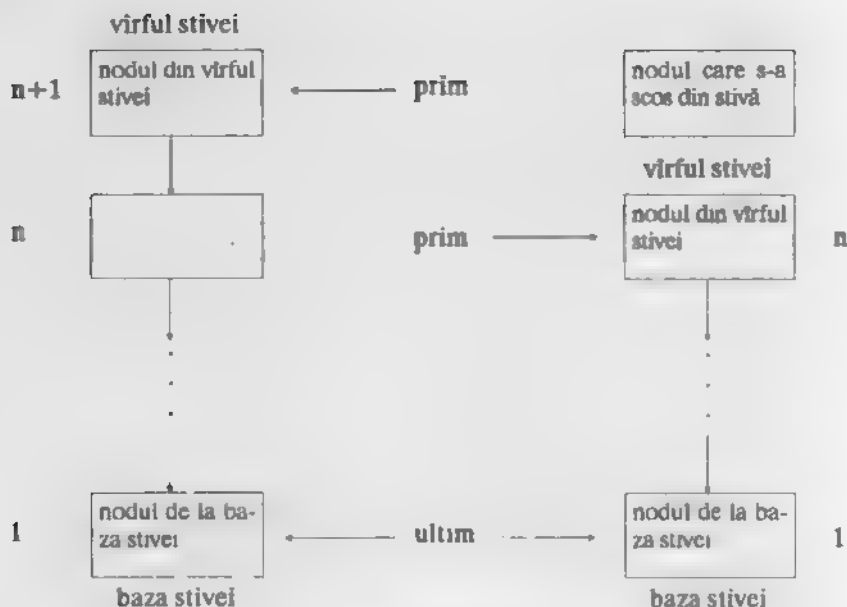
În sfîrșit, să amintim că vidarea stivei se realizează cu ajutorul funcției *sterglist* definită în paragraful 11.1.5.

În concluzie, o stivă se poate implementa printr-o listă simplu înlănțuită, pentru care se pot utiliza funcțiile:

- iniprim* - Realizează operația *push*.
- spn* - Realizează operația *pop*.
- sterglist* - Realizează operația *clear*.



Operația push (*iniprim*)



### Operația pop (spn)

Stivele pot fi implementate ca în exercițiul 7.1., folosind liste care la rîndul lor se implementează cu ajutorul tabloului de pointeri *tpnod* (vezi paragraful 11.1).

În acest caz, la baza stivei se află nodul spre care pointează elementul *tpnod[0]*, iar în vârful stivei se află nodul spre care pointează elementul *tpnod[tpnod-1]*.

În acest caz se vor utiliza funcțiile definite în paragraful 11.1, astfel:

- tpadauga* - Realizează operația *push*.
- tpsun* - Realizează operația *pop*.
- tpsterglist* - Realizează operația *clear*.

De obicei, în acest caz se mai definesc două operații asupra stivei:

*isempty* și *isfull*

Prima se definește printr-o funcție care returnează valoarea 1 dacă stiva este vidă și zero în caz contrar. Cea de a doua returnează valoarea 1 dacă stiva este plină și zero în caz contrar. Păstrînd denumirile de mai sus, aceste funcții se definesc simplu astfel:



```

typedef enum {false,true} Boolean;
Boolean isempty() /* returneaza true daca stiva este vida
                  si false altfel */
{
    extern int itpnod;
    return itpnod==0;
}

Boolean isfull() /* returneaza true daca stiva este plina
                 si false altfel */
{
    extern int itpnod;
    return itpnod >= MAX;
}

```

Un alt principiu de gestiune a listelor simplu înlanțuite este principiul FIFO (First In-First Out).

Conform acestui principiu, primul element introdus în listă este și primul care este scos din listă.

Despre o listă gestionată în acest fel se spune că formează o *coadă*.

Cele două capete ale listei simplu înlanțuite care implementează o coadă sînt și capetele cozii. Asupra cozilor se definesc trei operații, ca și asupra stivelor:

- a. pune un element în coadă;
- b. scoate un element din coadă;
- c. ștergerea (vidarea) unei cozi.

Pentru a respecta principiul FIFO, vom pune un element în coadă folosind funcția *adauga* și vom scoate un element din coadă folosind funcția *spn*. Deci, la un capăt al cozii se pun elemente în coadă, iar din celălalt capăt se scot elementele din coadă.

Ambele funcții, *adauga* și *spn* sînt funcții eficiente.

Ștergerea unei liste se realizează cu ajutorul funcției *sterglist*.

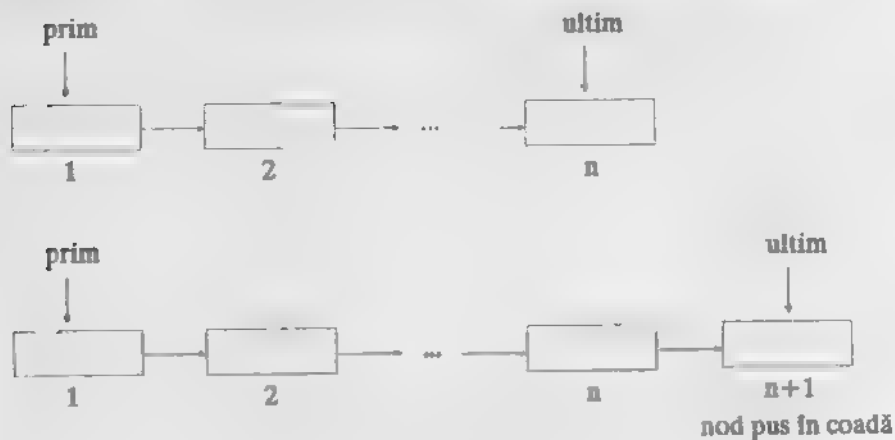
Primele două funcții sînt schematizate în figura următoare.

În concluzie, *coada* este o listă simplu înlanțuită pentru care se pot utiliza funcțiile:

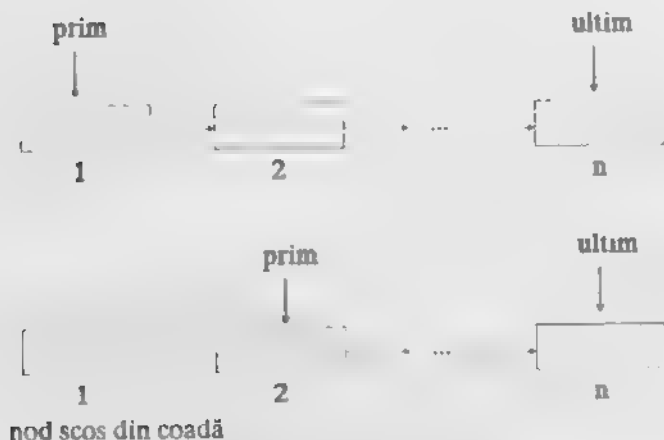
- |               |   |
|---------------|---|
| <i>adauga</i> | - Realizează operația de adăugare a unui nod la coadă.  |
| <i>spn</i>    | - Realizează operația de scoatere a unui nod din coadă. |

*stergist*

- Realizează ștergerea nodurilor existente în coadă.



### Operația de punere în coadă



### Operația de scoatere din coadă

Cozile definite ca mai sus corespund celor din viața de toate zilele și din această cauză ele se folosesc frecvent în probleme de simulare a fenomenelor reale.

Stivele se utilizează la descrierea proceselor recursive, inversarea ordinii elementelor unei mulțimi ordonate etc.

### Exerciții:

11.8 Într-o gară se consideră un tren de marfă ale cărui vagoane sînt inventariate într-o listă, în ordinea vagoanelor. Lista conține, pentru fiecare vagon, următoarele date:

- codul vagonului (9 cifre);
- codul conținutului vagonului (9 cifre);
- adresa expeditorului (4 cifre);
- adresa destinatarului (4 cifre).

Deoarece în gară se inversează poziția vagoanelor, se cere listarea datelor despre vagoanele respective în noua lor ordine.

În acest scop se crează o stivă în care se păstrează datele fiecărui vagon. Datele corespunzătoare unui vagon constituie un element al stivei, adică un nod al listei simplu înlanțuite. După ce datele au fost puse pe stivă, ele se scot de acolo și se listează. În acest mod se obține lista vagoanelor în ordine inversă celei inițiale.

Stiva este o listă simplu înlanțuită pe care o gestionăm folosind funcțiile:

*iniprim* și *spn*.

Aceste funcții sînt generale și ele sînt definite în paragrafele 11.1.3.1. și respectiv 11.1.4.1.

Funcția *iniprim* apelează funcția *uncnod* și *elibnod*, iar funcția *spn* numai funcția *elibnod*. Aceste două funcții (*uncnod* și *elibnod*) sînt specifice aplicației.

Funcția principală apelează repetat funcția *iniprim* pentru a pune datele pe stivă, apoi le scoate de pe stivă și le listează, pînă cînd aceasta devine vidă.

### PROGRAMUL BX18

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
```

```
typedef struct tnod {
    long cvag;
    long cmarfa;
    int exp;
```

```

    int dest;
    struct tnod *urm;
} TNOD;

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */

int incnod(TNOD *p) /* incarca un nod cu datele despre vagoane */
{
    char t[255];
    char er[] = "s-a tastat EOF in pozitie rea\n";
    long cod;
    int icod;

/* citeste cod vagon */
    for ( ; ; ) {
        printf("cod vagon: ");
        if(gets(t) == 0 )
            return -1; /* nu mai sint date */
        if(sscanf(t,"%ld",&cod)==1 && cod >= 0 &&
            cod <= 999999999)
            break;
        printf("cod vagon eronat\n");
    }
    p -> cvag = cod;

/* citeste cod marfa */
    for( ; ; ) {
        printf("cod marfa: ");
        if(gets(t) == 0 ) {
            printf(er);
            return 0;
        }
        if(sscanf(t,"%ld",&cod)==1 && cod >= 0 &&
            cod <= 999999999)
            break;
        printf("cod marfa eronat\n");
    }
    p -> cmarfa = cod;

/* citeste cod expeditor */
    if(pcit_int_lim("cod expeditor: ",0,9999,
        &icod) == 0 ) {

```

```

        printf(er);
        return 0;
    }
    p -> exp = icod;

/* citește cod destinatar */
    if(pcat_int_lim("cod destinatar: ",0,9999,
        &icod) == 0 ) {
        printf(er);
        return 0;
    }
    p -> dest = icod;
    return 1;
} /* sfîrșit incnod */

void elibnod(TNOD *p) /* eliberează nodul spre care
                        pointează p */
{
    free(p);
} /* sfîrșit elibnod */

TNOD *iniprim() /* inserează nodul curent înaintea primului nod
                 al listei - push */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n = sizeof(TNOD);
    if(((p = (TNOD *)malloc(n)) != 0 ) &&
        (incnod(p) == 1 )) {
        if(prim == 0 ) {
            prim = ultim = p;
            p -> urm = 0;
        }
        else {
            p -> urm = prim;
            prim = p;
        }
        return p;
    }
    if(p == 0 ) {
        printf("memorie insuficientă\n");
    }
}

```

```

        exit(1);
    }
    elibnod(p);
    return 0;
} /* sfirsit iniprim */

void spn() /* sterge primul nod din lista - pop */
{
    extern TNOD *prim, *ultim;
    TNOD *p;

    if(prim == 0 )
        return ;
    p = prim;
    prim = prim -> urm;
    elibnod(p);
    if(prim == 0 )
        ultim = 0;
} /* sfirsit spn */

TNOD *prim, *ultim;

main() /* listeaza inventarul vagoanelor in ordinea inversa citirii lor */
{
    prim = ultim = 0; /* la inceput stiva este vida */

    /* se creaza stiva apelind iniprim pina cind aceasta returneaza valoarea
    zero */
    while(iniprim() != 0)
        ;

    /* - se listeaza elementul din virful stivei si apoi se sterge din stiva;
    - se repeta pina cind stiva devine vida. */
    while(prim != 0 ) {
        printf("cod vagon: %ld\tcontinut: %ld\n",
            prim -> cvag, prim -> cmarfa);
        printf("expeditor: %d\tdestinatar: %d\n",
            prim -> exp, prim -> dest);
        spn();
    }
} /* sfirsit main */

```

11.9 Cozile se folosesc adesea la simularea fenomenelor reale pe calculator.

Un exemplu simplu de simulare cu ajutorul cozilor este descris mai jos.

Presupunem o agenție C.E.C. mică cu un singur ghișeu, care se deschide la ora 8. Agenția se închide la ora 16, dar publicul aflat la coadă este deservit în continuare. Cu alte cuvinte, după ora 16 publicul nu mai are voie să intre în agenție ca să se așeze la coada de la ghișeu. Întrucât s-a constatat că de obicei coada este destul de mare la ora închiderii, se ridică problema oportunității deschiderii a încă unui ghișeu la agenția respectivă. În acest scop se realizează o simulare pe calculator a situației existente care să stabilească o medie a orelor suplimentare efectuate zilnic pe o perioadă de un an.

Pentru a simula acest fenomen se au în vedere operațiile efectuate la ghișeu, după cum urmează:

Operație	Time de execuție
1 - depunere	5 minute
2 - restituire fără confirmare	7 minute
3 - depunere pe un carnet nou	10 minute
4 - restituire cu confirmare	20 minute
5 - lichidare	25 minute

Operațiile care se efectuează nu sînt uniform repartizate. În medie, operațiile 1 și 2 se execută cel mai frecvent iar operația 5 cel mai rar. În medie, s-a constatat că operațiile 3 și 4 se solicită triplu față de operația 5, iar operațiile 1 și 2 sînt de 7 ori mai solicitate decît operația 5.

Avînd în vedere acest fapt, considerăm o metodă de simulare a operațiilor care se execută la ghișeu bazată pe numere pseudoaleatoare. Dacă se ia în seamă operația 5, ca unitate, înseamnă că pentru operațiile 1 și 2 vom alege ponderea 7, iar pentru operațiile 3 și 4 ponderea 3:

Operație	Pondere
operația 1	7
operația 2	7
operația 3	3
operația 4	3
operația 5	1
<hr/>	
Total	21

Se vor genera numere pseudoaleatoare situate în intervalul [1,21], iar operația se definește astfel:

Fie  $r$  un număr pseudoaleator din intervalul [1,21]:

- dacă  $1 \leq r \leq 7$ , atunci se realizează operația 1;
- dacă  $8 \leq r \leq 14$ , atunci se realizează operația 2;
- dacă  $15 \leq r \leq 17$ , atunci se realizează operația 3;
- dacă  $18 \leq r \leq 20$ , atunci se realizează operația 4;
- dacă  $r = 21$ , atunci se realizează operația 5.

Numerele pseudoaleatoare pot fi generate folosind funcțiile *random* și *srand* din biblioteca limbajelor C și C++.

Funcția *random* are prototipul:

**int random(int n);**

Ea returnează un număr pseudoaleator aflat în intervalul [0,n) (număr natural mai mic decât  $n$ ).

Numerele pseudoaleatoare se generează printr-un proces de calcul iterativ. Acest proces pornește cu o valoare inițială care poate fi definită de programator apelând funcția *srand*. Această valoare inițială se numește sămînța șirului de numere pseudoaleatoare.

Funcția *srand* are prototipul:

**void srand(unsigned n);**

unde:

$n$  - Este valoarea la care se setează sămînța după apelul funcției *srand*.

Ambele funcții au prototipul în fișierul *stdlib.h*.

Programul de simulare a problemei indicate mai sus construiește o coadă de așteptare cu persoanele care sosesc la agenție în intervalul de timp indicat mai sus.

Apelând funcția *random* se determină operația solicitată de fiecare persoană aflată la coadă. Se scot elementele din coadă respectind principiul FIFO și procesul continuă pînă la ora 16 cînd se sistează operația de adăugare. Se determină timpul necesar pentru realizarea operațiilor solicitate de persoanele aflate la coadă.

Rămîne de precizat modul în care vin persoanele la agenție. Vom presupune că intervalul de timp între două persoane care vin la agenție este aleator și că acesta este de maximum 15 minute. În acest caz se poate apela



funcția *random* cu parametrul 15 și valoarea:

`random(15)+1`

va reprezenta intervalul la care sosește persoana următoare la agenție.

Programul folosește o variabilă globală *timpcrt* a cărei valoare este numărul minutelor scurse de la ora 8 și până în momentul în care a sosit ultima persoană la agenție.

Avem 8 ore de lucru la ghișeu, deci  $\text{timpcrt} \leq 8 \cdot 60 = 480$  minute.

O altă variabilă care numără minutele rezultate din deservirea persoanelor care s-au aflat la coadă la ghișeu este *timpghiseu*.

Valorile acestor variabile satisfac relația:

`timpghiseu <= timpcrt`

Cînd o persoană se așează la coadă, atunci *timpcrt* se mărește cu timpul dat de expresia:

`random(15)+1.`

Cînd o persoană se scoate din coadă (a fost deservită), *timpghiseu* se mărește cu timpul necesar pentru operația solicitată de persoana respectivă, operație care se definește cu ajutorul expresiei:

`random(21)+1.`

Deoarece *timpcrt* definește timpul curent (numărul de minute care s-au socotit începînd cu ora 8 și pînă în momentul în care ultima persoană s-a așezat la coadă), rezultă că persoana din față este deservită (scoasă din coadă) numai dacă:

`timpghiseu + timp necesar pentru operația ei <= timpcrt`

Altfel se adaugă o nouă persoană la coadă dacă  $\text{timpcrt} \leq 480$ .

De asemenea, persoana din fața cozii este deservită, dacă  $\text{timpcrt} > 480$ .

## PROGRAMUL BXI9

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <conio.h>
```

```
typedef struct tnod {
    int timpoperație;
    struct tnod *urm;
```

```

} TNOD;

#define MAXTIMP 480
#define INTERVAL 15

int incnod(TNOD *p)
/* - incarca nodul curent daca timpul curent nu depaseste pe cel admis si
   returneaza 1;
   - altfel returneaza -1. */
{
    extern int timpert;
    int r;

    /* determina timpul, in minute, la care soseste persoana la agentie */
    timpert += random(INTERVAL) + 1;
    if(timpert > MAXTIMP )
        return -1; /* agentie inchisa */

    /* determina operatia si pastreaza in nod timpul necesar operatiei
       respective */
    r = random(21) + 1;
    if(r <= 7 )
        r = 5;
    else
        if( r <= 14 )
            r = 7;
        else
            if( r <= 17 )
                r = 10;
            else
                if( r <= 20 )
                    r = 20;
                else
                    r = 25;
    p -> timpoperatie = r;
    return 1;
} /* sfirsit incnod */

void elibnod( TNOD *p)
/* elibereaza zona ocupata de nodul listei spre care pointeaza p */
{
    free(p);
} /* sfirsit elibnod */

```

```

#include "bxi3.cpp" /* adauga */

void spn() /* sterge primul nod al listei(cozii) */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if( prim == 0 )
        return;
    p = prim;
    prim = prim -> urm;
    elibnod(p);
    if(prim == 0 )
        ultim = 0;
} /* sfirsit spn */

TNOD *prim,*ultim;
int timpert;

main()
/* simuleaza coziile de la o agentie CEC pe o perioada de 1 an */
{
    int timpghiseu;
    int i;

    for(i=1;i <= 360;i++){
        srand(i*10); /* seteaza saminta sirului pseudoaleator */

        /* initializari la deschiderea agentiei */
        timpert = timpghiseu= 0;
        prim = ultim = 0; /* coada este vida */
        while( adauga() ) /* ultima persoana sosita la agentie
                           se pune la coada */

        /* - se deservesc persoanele aflate la coada;
           - trebuie ca:
              1 - prim != 0 (altfel nu exista coada);
              2 - timpghiseu <= timpert. */
            while ( prim != 0 &&
                    (timpghiseu <= timpert) ) {

        /* se deserveste prima persoana din coada */
        timpghiseu += prim -> timpoperatie;
    }
}

```

```

/* se elimina din coada persoana deservita */
    apn();
}

/* - se inchide agentia;
   - se deservesc in continuare persoanele aflate la coada la ghiseu. */
while( prim != 0) {
    timpghiseu += prim -> timpoperatie;
    apn();
}

/* se afiseaza timpul suplimentar in minute */
if(timpghiseu - 480 > 0){
    printf("timp peste ora 16: %d minute\n",
        timpghiseu - 480);

    if(i%22==0){
        printf("Actionati o tasta pentru a\
            continua\n");
        getch();
    }
}
}
} /* sfirsit main */

```

## 11.3. Listă circulară simplu înlănțuită

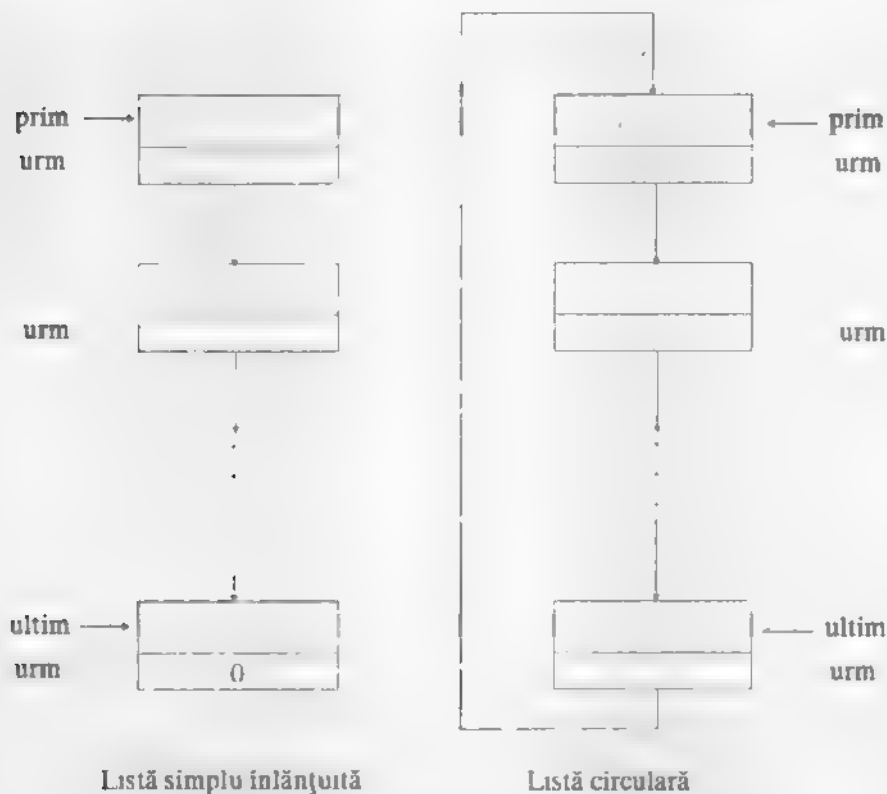
În paragraful 11.1 s-a definit lista simplu înlănțuită ca o mulțime ordonată de noduri, fiecare nod conținând un pointer spre un alt nod al listei, numit următorul nodului respectiv, exceptând un singur nod, care nu mai are următor. Acest nod, care nu mai are următor constituie un capăt al listei simplu înlănțuite. Tot în paragraful amintim mai sus, am convenit ca spre acest nod să poarte variabila *ultim*. De asemenea, lista simplu înlănțuită conține un nod care nu este următorul nici unui alt nod al ei. Acest nod constituie celălalt capăt al listei și spre el poartă variabila *prim*.

Pointerul prezent în fiecare nod al listei care definește ordinea nodurilor a fost numit *urm*. Conform celor spuse mai sus, *ultim -> urm=0*. Dacă într-o listă simplu înlănțuită schimbăm valoarea expresiei *ultim -> urm* făcând:

```
ultim -> urm=prim
```

atunci lista simplu înlanțuită devine o *listă simplu înlanțuită circulară*. În continuare prin *listă circulară* vom înțelege o listă circulară simplu înlanțuită.

Procesul de transformare a listei simplu înlanțuite în listă circulară este schematizat în figura de mai jos.



### Transformarea unei liste simplu înlanțuită în listă circulară

Într-o listă circulară toate nodurile sînt echivalente: fiecare nod are un următor și fiecare nod este următorul unui nod. Într-o astfel de listă nu mai sînt capete și de aceea nu mai sînt necesare variabilele *prim* și *ultim*. Gestiunea nodurilor listei circulare se realizează folosind o variabilă globală care pointează spre un nod oarecare al listei. Numim *ptrnod* această variabilă. Ea se declară astfel:

TNOD \*ptrmod;

unde:

**TNOD** - Este tipul comun nodurilor listei.

Asupra listelor circulare se definesc aceleași operații ca și asupra listelor simplu înlanțuite.

Listele circulare au o serie de aplicații dintre care amintim:

- operații cu numere întregi care au un număr mare de cifre;
- operații asupra polinoamelor de una sau mai multe variabile;
- alocarea dinamică a memoriei.

Menționăm că funcțiile *malloc*, *free*, precum și celelalte utilizate la gestiunea dinamică a memoriei utilizează o zonă de memorie organizată ca o listă circulară.

Memoria liberă este o listă circulară de noduri, fiecare cu o zonă de memorie liberă. Un nod conține dimensiunea lui, un pointer spre nodul următor din listă și spațiul liber care se pune la dispoziția utilizatorului. Alocarea se face astfel:

La un apel a lui *malloc* se parcurg nodurile listei până când se găsește o zonă de memorie de dimensiune cel puțin egală cu cea cerută la apel. Dacă zona este mai mare, atunci ea se divide și partea neutilizată se înlanțuie cu celelalte blocuri ale listei. Când nu se găsește o zonă de memorie corespunzătoare, atunci se încearcă obținerea ei printr-un apel la sistemul de operare.

Eliberarea unei zone de memorie prin intermediul funcției *free* va înlanțui zona respectivă la lista circulară. Dacă două noduri cu zone de memorie liberă ocupă zone contigue, atunci ele se concatenează formind o singură zonă liberă de dimensiune egală cu suma dimensiunilor lor. De aceea, se recomandă utilizarea funcției *free* pentru a elibera zone de memorie de îndată ce nu mai este nevoie de ele. În cazul acesta se poate preîntîmpina divizarea excesivă a zonei gestionată dinamic prin funcțiile de felul lui *malloc* și *free*.

Mai jos definim funcții pentru a realiza operațiile de bază asupra listelor circulare.

### 11.3.1. Crearea unei liste circulare

La crearea unei liste circulare, ca și la crearea unei liste simplu înlanțuite, se utilizează funcțiile *incnod* și *elibnod*.

Prima se apelează pentru a încărca datele curente într-un nod al listei, iar cea de a doua pentru a elibera zonele de memorie alocate pentru un nod.

Amintim că funcția *incnod* returnează:

- 0 - La eroare.
- 1 - La încărcarea normală.
- 1 - Nu mai sînt date de încărcat în nod.

Funcția de creare a listei circulare returnează:

- 0 - La eroare.
- 1 - La crearea fără erori a listei.

```
int ccrelist() /* - creaza o lista circulara,
                - returneaza:
                  0 - la eroare;
                  -1 - la creare normala. */
{
    extern TNOD *ptrnod;
    int i, n;
    TNOD *p;

    n=sizeof(TNOD);
    ptrnod=0; /* lista este vida la inceput */
    while((p=(TNOD *)malloc(n))!=0) {
        (i=incnod(p))==1)

/* s-a rezervat zona pentru nod si s-au incarcat date in zona respectiva */
        if(ptrnod==0) { /* lista vida */
            ptrnod=p;
            ptrnod -> urm=p;
        }
        else { /* nodul curent se insereaza dupa cel spre care pointeaza
                ptrnod */
            p -> urm=ptrnod -> urm;
            ptrnod -> urm=p;
            ptrnod=p; /* ptrnod pointeaza spre ultimul nod inserat in
                        lista */
        } /* sfirsit else */

/* sfirsit while: p=0 sau incnod nu a returnat valoarea 1 */
        if(p==0) { /* nu s-a rezervat zona pentru nod */
```

```

    printf("memorie insuficienta\n");
    exit(1);
}
/* - s-a rezervat zona pentru nod dar incnod nu a reusit sa incarce date in
   zona respectiva returnind o valoare diferita de unu, deci zero sau -1;
   - valoarea returnata de incnod va fi returnata si de functia
   ccrelist. */

elibnod(p); /*elibereaza zona de memorie rezervata pentru nod
             si care n-a mai fost incarcata cu date */
return i; /* i are ca valoare valoarea returnata de incnod */
}

```

### Observație:

Funcția *ccrelist* a fost scrisă mai compact decît funcția *crelist* utilizînd expresia:

```
((p=(TNOD *)malloc(n))!=0)&&((i=incnod(p))==1)
```

care se evaluează astfel:

Se apelează funcția *malloc* pentru a rezerva *n* octeți în memoria *heap*.

În cazul în care se poate rezerva o zonă de *n* octeți, lui *p* i se atribuie o valoare diferită de zero și deci primul operand al operatorului && are valoarea adevărat. În acest caz se evaluează operandul al doilea al operatorului &&, care apelează funcția *incnod* pentru a încărca datele curente în zona a cărei adresă de început este valoarea lui *p*.

În cazul în care nu se pot rezerva *n* octeți în memoria *heap*, lui *p* i se atribuie valoarea zero și deci primul operand al operatorului && este fals.

În acest caz întreaga expresie este falsă și deci nu se mai evaluează cel de al doilea operand al operatorului &&.

De asemenea, expresia este falsă și în cazul în care lui *p* i s-a atribuit o valoare diferită de zero, dar *incnod* a returnat o valoare diferită de unu.

### 11.3.2. Accesul la un nod al unei liste circulare

Ca în cazul listelor simplu înălțuite și în cazul listelor circulare putem căuta un nod după o cheie sau mai multe chei. În cazul listelor circulare, căutarea va începe cu nodul spre care pointează variabila globală *ptrnod*.

Mai jos, se definește funcția *ccnci* care este analogă funcției *cnci* definită



în cazul listelor simplu înlanțuite. Ea caută un nod după o cheie numerică și returnează una din valorile:

- pointerul spre nodul căutat;
- 0 dacă nu există un astfel de nod.

Amintim că în acest caz tipul nodurilor listei se declară astfel:

```
typedef struct tnod {  
    declaratii  
    int cheie;  
    declaratii  
} TNOD;
```

```
TNOD *cenci(int c)  
/* - cauta nodul de cheie=c;  
   - returneaza pointerul spre nodul respectiv sau 0 daca nu exista  
   un astfel de nod. */  
{  
    extern TNOD *ptrnod;  
    TNOD *p;  
  
    p=ptrnod;  
    if(p==0) /* lista vida */  
        return 0;  
    do{  
        if(p -> cheie==c)  
            return p;  
        p=p -> urm;  
    }while(p!=ptrnod);  
    return 0;  
}
```

### 11.3.3. Inserarea unui nod într-o listă circulară

Considerăm două operații de inserare a unui nod într-o listă simplu înlanțuită:

- inserarea unui nod înaintea unuia precizat printr-o cheie numerică de tip *int*;
- inserarea unui nod după unul precizat printr-o cheie numerică de tip *int*.

Cititorul poate defini și alte funcții de inserare similare cu cele prezentate mai jos.

### 11.3.3.1. Inserarea unui nod înaintea unuia precizat printr-o cheie de tip int

Nodurile listei au tipul TNOD indicat în paragraful 11.3.2.

```
TNOD *cinici(int c)
/* - inserarea unui nod într-o listă circulară înaintea unui nod precizat
   printr-o cheie de tip int;
   - returnează pointerul spre nodul inserat sau zero dacă inserarea nu
   are loc. */
{
    extern TNOD *ptrnod;
    TNOD *p,*q,*q1;
    int n;

    if(ptrnod==0)
        return 0; /* lista vidă */
    q=ptrnod;
    do{
        q1=q;
        q=q->urm;
        if(q->cheie==c)
            break; /* s-a găsit nodul înaintea căruia se va face
                    inserarea */
    }while(q1!=ptrnod);
    if(q->cheie!=c){
        printf("nu există un nod de cheie=%d\n",c);
        return 0; /* nu s-a făcut nici o inserare */
    }

    /* rezerva zona pentru nod și încarcă datele în nodul respectiv */
    n=sizeof(TNOD);
    if((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        q1->urm=p;
        p->urm=q;
        return p;
    }
    if(p==0){
        printf("memorie insuficientă\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

### 11.3.3.2. Inserarea unui nod după un nod precizat printr-o cheie de tip int

Tipul TNOD se definește ca în paragraful 11.3.2.

```
TNOD *cinci(int c)
/* insereaza un nod intr-o lista circulara dupa un nod precizat printr-o
   cheie de tip int */
{
    extern TNOD *ptrnod;
    TNOD *p,*q;
    int n;

    if(ptrnod==0)
        return 0; /* lista vida */
    q=ptrnod;
    do{
        if(q -> cheie==c)
            break;
        q=q -> urm
    }while(q!=ptrnod);
    if(q -> cheie !=c){
        printf("nu exista un nod de cheie =%d\n",c);
        return 0;
    }

    /* se face inserarea dupa nodul spre care pointeaza q */
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        p -> urm=q -> urm;
        q -> urm=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

### 11.3.4. Ștergerea unui nod dintr-o listă circulară

Dăm mai jos o funcție care permite ștergerea dintr-o listă circulară a unui nod precizat printr-o cheie de tip *int*.

În cazul în care variabila *ptrnod* pointează chiar spre nodul care se șterge, convenim ca *ptrnod* să poarte spre nodul precedent celui șters, dacă lista n-a devenit vidă. În acest ultim caz, lui *ptrnod* i se atribuie valoarea zero.

Tipul *TNOD* se definește ca în cazul paragrafului precedent.

După modelul funcției de mai jos se pot defini și alte funcții pentru a șterge noduri dintr-o listă circulară.

```
void canci(int c) /* șterge nodul pentru care cheie=c */
{
    extern TNOD *ptrnod;
    TNOD *p,*pl;

    if(ptrnod==0)
        return; /* lista vidă */
    p=ptrnod;
    do{
        pl=p;
        p=p->urm;
        if(p->cheie==c)
            break;
    }while(pl!=ptrnod);
    if(p->cheie!=c){
        printf("lista nu contine un nod de\
              cheie=%d\n",c);
        return;
    }
    if(p==p->urm){ /* lista are un singur nod */
        ptrnod=0;
    }
    else{
        pl->urm=p->urm;
        if(p==ptrnod) /* se șterge nodul spre care pointeaza
                        ptrnod */
            ptrnod=pl;
    }
    elibnod(p);
}
```

### 11.3.5. Ștergerea unei liste circulare

```
void csterglist() /* șterge o lista circulară */
{
    extern TNOD *ptrnod;
    TNOD *p,*pl;

    if((p=ptrnod)==0)
        return; /* lista vidă */
    do{
        pl=p;
        p=p -> urm;
        elibnod(pl);
    }while(pl!=ptrnod);
    ptrnod=0;
}
```

#### Exerciții:

11.10 Se consideră tipul TNOD declarat ca mai jos:

```
typedef struct tnod {
    char *cuv;
    struct tnod *urm;
} TNOD;
```

Acest tip se utilizează în toate exercițiile de la acest paragraf.

Mai jos, definim funcția care încarcă datele într-un nod de tipul TNOD.

#### FUNCȚIA BXI10

```
int incnod(TNOD *p)
/*- încarcă datele în nodul spre care pointează p;
- returnează:
    -1 la întâlnirea sfârșitului de fișier;
    1 altfel. */
{
    char t[255];

    p -> cuv = 0; /* inițializarea cu pointerul nul */
    printf("tastati pe un rînd cuvîntul curent\n");
    if(gets(t) == 0 )
        return -1; /* s-a tastat EOF */
}
```

```

/* rezerva zona pentru rindul citit */
if((p -> cuv =
    (char *)malloc(strlen(t) + 1)) == 0 ) {
    printf("memorie insuficienta\n");
    exit(1);
}

/* pastreaza rindul citit in memoria heap */
strcpy(p -> cuv , t);
return 1;
}

```

- 11.11 Să se definească funcția *elibnod* care eliberează zonele de memorie ocupate de un nod de tip TNOD.

### **FUNCȚIA BX111**

```

void elibnod(TNOD *p)
/* elibereaza zonele de memorie ocupate de nodul spre care
   pointeaza p */
{
    free( p -> cuv);
    free(p);
}

```

- 11.12 Să se scrie funcția *occrelist* care crează o listă circulară ale cărei noduri sînt de tipul TNOD definit în exercițiul 11.10.

### **FUNCȚIA BX112**

```

int occrelist()
/* - creaza o lista circulara;
   - returneaza:
       0 la eroare;
       -1 altfel. */
{
    extern TNOD *ptrnod;
    int i,n;
    TNOD *p;

    n = sizeof(TNOD);
    ptrnod = 0;
    while((p = (TNOD *)malloc(n)) != 0 ) &&

```

```

((i = incnod(p)) == 1 ))
    if(ptrnod == 0 ) {
        ptrnod = p;
        ptrnod -> urm = p;
    }
    else {
        p -> urm = ptrnod -> urm;
        ptrnod -> urm = p;
        ptrnod = p;
    }
}
if( p == 0 ) {
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return i;
}

```

- 11.13 Să se scrie o funcție care caută un nod al listei circulare create prin funcția *ccrelist*, nod pentru care pointerul *cuv* pointează spre un șir de caractere dat.

Funcția returnează pointerul spre nodul respectiv sau zero dacă nu există un astfel de nod.

## FUNCȚIA BX113

```

TNOD *ccncs( char *c)
/* - cauta nodul pentru care cuv si c pointeaza spre
   același șir de caractere;
   - returneaza:
       pointerul spre nodul respectiv sau zero daca nu exista un
       astfel de nod. */
{
    extern TNOD *ptrnod;
    TNOD *p;

    p = ptrnod;
    if(ptrnod == 0)
        return 0; /* lista vida */
    do {
        if(strcmp( p -> cuv, c) == 0)
            return p;
        p = p -> urm;
    }
}

```

```

    } while ( p != ptrnod);
    return 0;
}

```

11.14 Fie lista circulară creată cu ajutorul funcției *ccrelist* definită în exercițiul 11.12., fie *pnod* pointerul spre un nod al listei circulare pentru care:

*pnod* -> *cuv*

poartează spre un șir dat și  $n > 1$  un întreg de tip *int*. Se cere nodul din listă obținut în urma eliminării nodurilor din listă în felul următor:

1. Se pornește cu nodul imediat următor nodului care conține pointerul spre șirul dat și se elimină din listă al  $n$ -lea nod care urmează după acest nod.
2. Se execută pasul 1 continuând cu nodul imediat următor celui șters, până când lista se reduce la un singur nod.

Nodul la care s-a redus lista este cel căutat

Această problemă se dă adesea ca exemplu pentru utilizarea listelor circulare. O variantă a acestei probleme este așa numita problemă a lui *Josephus*. Ea se formulează ca mai jos.

O cetate este apărată de un număr de soldați care își dau seama că au nevoie de ajutoare pentru a rezista în fața dușmanului care îi atacă. Se pune problema de a alege pe unul dintre ei care să plece după ajutor.

Alegerea se face așezând soldații în cerc și trăgând la sorți numele soldatului de la care să înceapă numărătoarea. De asemenea, se trage la sorți un număr întreg  $n > 1$ . Se numără, în sensul acelor ceasornicului, începând cu soldatul următor celui al cărui nume a fost tras la sorți și al  $n$ -lea soldat este scos din cerc. Se continuă numărătoarea în același fel începând cu soldatul care urmează după cel scos din cerc. În felul acesta, după un număr finit de pași, cercul se reduce la un singur soldat cărui îi revine sarcina să plece după ajutoare.

Problema lui *Josephus* este o variantă a formulării descrise prin punctele 1-2 de mai sus, dacă se consideră că pointerul *cuv* poartează spre numele unui soldat. La punctul 1 se precizează că se pornește cu un nod care urmează imediat nodului care conține pointerul spre un șir dat. Acest șir dat, este chiar numele soldatului tras la sorți, care apare în problema lui *Josephus*.

Programul de mai jos rezolvă această problemă conform următorilor pași:



- Citește numărul  $n$  indicat în formularea problemei.
- Citește un șir de caractere care definește nodul de la care începe numărătoarea.
- Crează lista circulară care trebuie să conțină un nod pentru care *cuv* pointează spre un șir identic cu cel citit la punctul b.
- Se elimină nodurile listei circulare conform pașilor 1 și 2 indicați mai sus.
- Se afișează cuvântul din nodul rămas în listă.

## PROGRAMUL BX114

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <alloc.h>
```

```
typedef struct tnod {
    char *cuv;
    struct tnod *urm;
} TNOD;
```

```
#include "bviii2.cpp" /* pcit_int    */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bx110.cpp"  /* incnod    */
#include "bx111.cpp"  /* elibnod    */
#include "bx112.cpp"  /* ccrelist   */
#include "bx113.cpp"  /* ccncs     */
```

```
#define MAXN 1000
TNOD *ptrnod;
```

```
main() /* - creeaza o lista circulara si elimina nodurile ei pornind de la
        un nod dat si eliminand tot al n-lea nod pina cind lista se
        reduce la un singur nod;
        - in final se listeaza sirul spre care pointeaza cuv al nodului
        la care s-a redus lista. */
```

```
{
    char t[255];
    int i,n;
    char er[]="s-a tastat EOF\n";
    TNOD *p,*pl;
```

```

/* citește pe n */
if( pcit_int_lim("n= ",2,MAXN,&n) == 0 ) {
    printf(er);
    exit(1);
}

/* citește un sir de caractere care va defini nodul din lista pentru pornirea
numararii nodurilor */
printf("sirul pentru pornirea numararii\n");
if( gets(t) == 0 ) {
    printf(er);
    exit(1);
}

/* creează lista circulară */
printf("tastati sirurile care intra in\
    compunerea listei\n");
printf("cite un sir pe un rind\n");
printf("la sfirsit se tasteaza Ctrl-Z\n");
ccreolist();

/* se determină nodul pentru care cuv pointează spre un sir identic cu cel
pastrat in tabloul t */
if((p = concs(t)) == 0 ) {

/* nu exista un nod pentru care sirul spre care pointează cuv sa coincidă
cu cel pastrat in t */
    printf("nu se poate determina nodul de la\
        care sa se înceapă numărarea\n");
    exit(1);
}

/* elimină nodurile din lista pînă se ajunge la un singur nod */
p=p ->urm; /* număratoarea începe cu nodul următor celui pentru
        care cuv pointează spre un sir identic cu cel pastrat
        in t */
while (ptrnod != ptrnod -> urm ) {

/* lista conține mai mult de un nod */

/* se caută al n-lea nod începînd cu cel spre care pointează p */
    for( i=1; i<n; i++) {
        pl = p;

```

```

    }
    p = p -> urm;
}

/* - se sterge nodul spre care pointeaza p;
   - p1 pointeaza spre nodul precedent nodului spre care pointeaza p. */
p1 -> urm = p -> urm;
if (ptrnod == p)
    ptrnod = p1;
free( p -> cuv );
free(p);
p = p1 -> urm; /* numaratoarea continua incepind cu
                nodul urmator celui sters */
}

/* lista s-a redus la un singur nod */
printf("sirul cautat: \n");
printf("%s\n", ptrnod -> cuv );
}

```

## 11.4. Listă dublu înlănțuită

Atît listele simplu înlănțuite, cit și listele circulare, conduc adesea la parcurgeri neefective ale lor. De exemplu, ștergerea ultimului nod al unei liste simplu înlănțuite implică parcurgerea listei respective (vezi funcția *sun*).

Astfel de parcurgeri pot fi uneori evitate folosind liste dublu înlănțuite. Într-o astfel de listă fiecare nod conține doi pointeri: unul spre nodul următor și unul spre nodul precedent.

În paragraful de față, vom presupune că nodurile listelor dublu înlănțuite au tipul definit ca mai jos:

```

typedef struct tnod {
    declarații
    struct tnod *prec;
    struct tnod *urm;
}TNOD;

```

Pentru a gestiona o listă dublu înlănțuită vom utiliza variabilele globale *prim* și *ultim*, ca în cazul listelor simplu înlănțuite. Variabila *prim* pointează spre nodul pentru care *prim*-> *prec*=0. Pentru restul nodurilor, pointerul *prec* al unui nod pointează spre nodul precedent al listei.

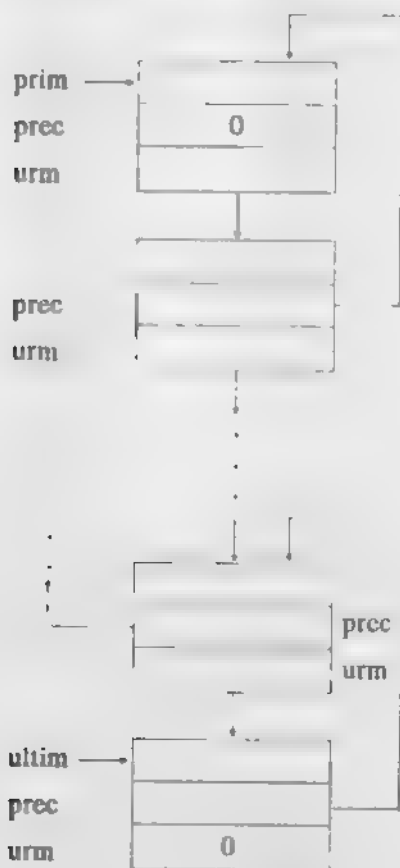
Variabila *ultim* pointează spre un nod pentru care *ultim* -> *urm*=0.

Pentru restul nodurilor, pointerul *urm* al unui nod pointează spre nodul următor al listei.

În concluzie, fiecare nod al listei are un nod precedent definit prin pointerul *prec* și un următor definit prin pointerul *urm*.

O excepție de la această regulă o constituie nodurile spre care pointează variabilele *prim* și *ultim*. Nodul spre care pointează *prim* nu are precedent, iar nodul spre care pointează *ultim* nu are următor. Aceste noduri constituie capetele listei dublu înălănțuite.

În figura de mai jos se dă o schemă pentru listele dublu înălănțuite.



**Listă dublu înălănțuită**

În legătură cu listele dublu înlanțuite se pot defini aceleași operații ca și în cazul listelor simplu înlanțuite:

- crearea unei liste dublu înlanțuite;
- accesul la un element al unei liste dublu înlanțuite;
- inserarea unui nod într-o listă dublu înlanțuită;
- ștergerea unui nod dintr-o listă dublu înlanțuită;
- ștergerea unei liste dublu înlanțuite.

Operațiile de la punctele *b* și *e* se realizează ca în cazul listelor simplu înlanțuite și de aceea ele nu vor mai fi considerate în paragrafele care urmează.

### 11.4.1. Crearea unei liste dublu înlanțuite

Definim mai jos funcția *dcrelist* utilizată pentru a crea o listă dublu înlanțuită. Ea este analogă cu funcția *crelist* definită în paragraful 11.1.1.

Funcțiile *incnod* și *elibnod* au aceeași semnificație și utilizare ca în cazul funcției *crelist*.

```
int dcrelist() /* - creaza o lista dublu inlantuita;
                - returneaza:
                  0 - la eroare;
                  -1 - creare normala. */
{
    extern TNOD *prim,*ultim;
    int i,n;
    TNOD *p;

    n=sizeof(TNOD);
    prim=ultim=0;
    while(((p=(TNOD *)malloc(n))!=0)&&
           ((i=incnod(p))==1))
        if(prim==0){
            prim=ultim=p;
            p -> prec=p -> urm=0;
        }
        else{
            ultim -> urm=p;
            p -> prec=ultim;
            p -> urm=0;
            ultim=p;
        }
    }
```

```

    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return i;
}

```

## 11.4.2. Inserarea unui nod într-o listă dublu înlănțuită

Într-o listă dublu înlănțuită se pot face inserări în diferite poziții. În acest paragraf vom considera câteva posibilități ca în cazul listelor simplu înlănțuite:

- inserare înaintea primului nod al listei (nodul spre care pointează variabila *prim* este primul nod al listei);
- inserare înaintea unui nod precizat printr-o cheie,
- inserare după un nod precizat printr-o cheie,
- inserare după ultimul nod al listei (nodul spre care pointează variabila *ultim* este ultimul nod al listei).

### 11.4.2.1. Inserarea unui nod într-o listă dublu înlănțuită înaintea primului nod al ei

```

TNOD *diniprim() /* - insereaza nodul curent inaintea primului
                    nod al listei;
                    - returneaza pointerul spre nodul inserat. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        if(prim==0){
            prim=ultim=p;
            p -> prec = p -> urm=0;
        }
        else{
            p -> urm = prim;
            p -> prec=0;
            prim -> prec = p;
        }
    }
}

```

```

        prim=p;
    }
    return p;
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

#### 11.4.2.2. Inserarea unui nod într-o listă dublu înălțuită înaintea unui nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*. Tipul TNOD se declară astfel:

```

typedef struct tnod {
    declaratii
    int cheie;
    declaratii
    struct tnod *prec;
    struct tnod *urm;
} TNOD;

```

Funcția de inserare apelează funcția *dcnci* care caută într-o listă dublu înălțuită un nod de cheie dată. Această funcție este identică cu funcția *cnci* definită în paragraful 11.1.2.

```

TNOD *dcnci(int c)
/* - cauta un nod al listei pentru care cheie=c;
   - returneaza pointerul spre nodul determinat in acest fel sau zero daca
      nu exista nici un nod pentru care cheie=c. */
{
    extern TNOD *prim;
    TNOD *p;

    for(p=prim;p;p=p->urm)
        if(p->cheie==c)
            return p;
    return 0;
}

```

```

TNOD *dinici(int c)
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numerica;

```

```

- returneaza pointerul spre nodul inserat sau zero daca nu are loc
  inserarea. */
{
    extern TNOD *prim;
    TNOD *p,*q;
    int n;

    /* cauta nodul pentru care cheie=c */
    if((q=dnci(c))==0){
        printf("nu exista in lista un nod de\
              cheie=%d\n",c);
        return 0;
    }
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        p -> prec=q -> prec;
        p -> urm=q;
        if(q -> prec!=0)

    /* precedentul lui q are ca urmator nodul inserat */
        q -> prec -> urm=p;
        q -> prec=p;
        if(prim==q)

    /* s-a inserat inaintea primului nod */
        prim=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

#### 11.4.2.3. Inserarea unui nod într-o listă dublu înlănțuită după unul precizat printr-o cheie

Vom presupune că, cheia este de tip *int*, iar tipul *TNOD* este cel declarat în paragraful precedent. De asemenea, pentru a localiza nodul de cheie precizată, se va utiliza funcția *dnci* definită în același paragraf.



```

TNOD *dindci(int c)
/* - insereaza un nod dupa unul precizat printr-o cheie numerica;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu
   are loc. */
{
    extern TNOD *ultim;
    TNOD *p,*q;
    int n;

    if((q=dnci(c))==0){
        printf("nu exista nodul de cheie=%d\n",c);
        return 0;
    }
    n=sizeof(TNOD);
    if((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        p -> prec=q;
        p -> urm=q -> urm;
        if(q -> urm!=0)

/* urmatorul lui q are ca si precedent nodul inserat */
        q -> urm -> prec=p;
        q -> urm=p;
        if(ultim==q)

/* s-a inserat dupa ultimul nod */
        ultim=p;
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

#### 11.4.2.4. Inserarea unui nod într-o listă dublu înălănțuită după ultimul nod (adăugarea unui nod la o listă dublu înălănțuită)

În acest caz tipul nodurilor se declară astfel:

```

typedef struct tnod {
    declaratu
    struct tnod *prec;

```

```

    struct tnod *urm;
} TNOD;

```

nefiind necesară prezența unei chei.

```

TNOD *adauga()
/* - adauga un nod la o lista dublu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se
   realizeaza inserarea. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
        if(prim==0){
            prim=ultim=p;
            p -> prec=p -> urm=0;
        }
        else{
            ultim -> urm=p;
            p -> prec = ultim;
            p -> urm=0;
            ultim=p;
        }
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}

```

### 11.4.3. Ștergerea unui nod dintr-o listă dublu înălănțuită

Dintr-o listă dublu înălănțuită se pot șterge noduri. În cele ce urmează vom avea în vedere următoarele cazuri:

- ștergerea primului nod al unei liste dublu înălănțuite;
- ștergerea unui nod precizat printr-o cheie;
- ștergerea ultimului nod al unei liste dublu înălănțuite.

#### 11.4.3.1. Ștergerea primului nod al unei liste dublu înălțuite

```
void dspn() /* șterge primul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if(prim==0)
        return;
    p=prim;
    prim=prim -> urm;
    elibnod(p);
    if(prim==0)
        ultim=0; /* lista a devenit vida */
    else
        prim -> prec=0;
}
```

#### 11.4.3.2. Ștergerea unui nod dintr-o listă dublu înălțuită precizat printr-o cheie

Vom considera că, cheia este de tip *int*. În acest caz tipul TNOD se declară ca în paragraful 11.4.2.2.

Funcția de față apelează funcția *dcnci*, definită în același paragraf, pentru a localiza nodul care urmează a fi șters.

```
void danci(int c) /* șterge nodul de cheie=c */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if(prim==0) /* lista vida */
        return;
    if((p=dcnci(c))==0){
        printf("lista nu contine nodul\nde cheie = %d\n",c);
        return;
    }
    if(prim==p&&ultim==p){
        /* lista are un singur nod; devine vida */
        prim=ultim=0;
        elibnod(p);
    }
}
```

```

        return;
    }
    if (prim==p) { /* se sterge primul nod din lista */
        prim=prim -> urm;
        prim -> prec=0;
        elibnod(p);
        return;
    }
    if (ultim==p) { /* se sterge ultimul nod */
        ultim=ultim -> prec;
        ultim -> urm=0;
        elibnod(p);
        return;
    }

    /* se sterge un nod diferit de capete */

    /* urmatorul nodului care se sterge are ca precedent, precedentul nodului
       care se sterge */
    p -> urm -> prec=p -> prec;

    /* precedentul nodului care se sterge are ca urmator, urmatorul nodului
       care se sterge */
    p -> prec -> urm=p -> urm;
    elibnod(p);
}

```

#### 11.4.3.3. Ștergerea ultimului nod al unei liste dublu înălănțuite

În acest caz tipul TNOD nu necesită prezența unei chei. Se poate utiliza declarația indicată în paragraful 11.4.2.4.

```

void dsun() /* sterge ultimul nod din lista */
{
    extern TNOD *prim,*ultim;
    TNOD *p;

    if (prim==0)
        return;
    p=ultim;
    ultim=ultim -> prec;
    if (ultim==0)
        prim=0; /* lista devine vida */
    else

```

```

        ultim -> urm=0;
    elibnod(p);
}

```

### Observații:

1. Funcția *dsun*, spre deosebire de funcția *sun* relativă la listele simplu înlanțuite, devine eficientă deoarece ea nu mai necesită parcurgerea nodurilor listei.
2. Funcțiile de inserare înainte și după un nod precizat printr-o cheie numerică întreagă sînt mai simple în cazul listelor dublu înlanțuite în comparație cu analogele lor pentru liste simplu înlanțuite deoarece ele folosesc funcția *dcnci* pentru localizarea nodului în raport cu care se face inserarea.
3. Funcția de ștergere a unui nod precizat printr-o cheie numerică întreagă este mai simplă în cazul listelor dublu înlanțuite în comparație cu funcția corespunzătoare pentru liste simplu înlanțuite deoarece ea folosește funcția *dcnci* pentru localizarea nodului care se șterge.
4. O listă dublu înlanțuită devine o listă circulară dublu înlanțuită dacă se fac atribuiri:

```

    ultim -> urm=prim;

```

și

```

    prim -> prec=ultim.

```

Pentru a gestiona o astfel de listă nu mai sînt necesare variabilele *prim* și *ultim*, lista ne mai avînd capete. În locul lor se utilizează un pointer spre un nod arbitrar al listei, ca în cazul listelor circulare simplu înlanțuite.

În legătură cu listele circulare dublu înlanțuite se ridică aceleași probleme ca și în cazul listelor circulare simplu înlanțuite.

Propunem cititorului să construiască funcții pentru a realiza operațiile de bază asupra listelor circulare dublu înlanțuite:

- creare;
- acces la un nod de cheie dată;
- inserări de noduri;
- ștergeri de noduri;
- ștergerea listei.

## Exerciții:

11.15 Se consideră tipul utilizator:

```
typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD
```

Să se scrie funcția *dadauga*, care permite adăugarea la o listă dublu înlanțuită a unui nod de tipul TNOD definit ca mai sus.

Se observă că funcția *dadauga* definită în paragraful 11.4.2.4, este dependentă numai de componentele *prec* și *urm* din TNOD. De aceea, funcția *dadauga* de mai jos este identică cu cea din paragraful 11.4.2.4.

## FUNCȚIA BX115

```
TNOD *dadauga()
/* - adauga un nod la o lista dublu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se
   realizeaza inserarea. */
{
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;

    n = sizeof(TNOD);
    if(((p = (TNOD *)malloc(n)) != 0) &&
        (incnod(p) == 1)) {
        if( prim == 0 ) {
            prim = ultim = p;
            p -> prec = p -> urm = 0;
        }
        else {
            ultim -> urm = p;
            p -> prec = ultim;
            p -> urm = 0;
            ultim = p;
        }
        return p;
    }
    if( p == 0 ) {
```

```

    printf("memorie insuficienta\n");
    exit(1);
}
elibnod(p);
return 0;
}

```

- 11.16 Să se scrie o funcție care șterge ultimul nod al unei liste dublu înlanțuite ale cărei noduri au tipul TNOD definit în exercițiul precedent.

Această funcție este definită în paragraful 11.4.3.3.

### FUNCȚIA BXI16

```

void dsun() /* șterge ultimul nod din lista */
{
    extern TNOD *prim, *ultim;
    TNOD *p;

    if(prim == 0 )
        return;
    p = ultim;
    ultim = ultim -> prec;
    if(ultim == 0 )
        prim = 0;
    else
        ultim -> urm = 0;
    elibnod(p);
}

```

- 11.17 Să se scrie un program care citește cuvintele dintr-un text și afișează numărul de apariții al fiecărui cuvânt din textul respectiv.

Această problemă a mai fost rezolvată în exercițiile 10.50 și 11.6. În exercițiul 11.6 s-a creat o listă simplu înlanțuită ale cărei noduri conțin pointeri spre cuvintele diferite din text, precum și frecvența de apariție a acestora în text.

În cazul de față, se utilizează o listă dublu înlanțuită. Prin aceasta, eficiența programului este mai mare deoarece funcția *dsun*, care șterge ultimul nod dintr-o listă dublu înlanțuită, este mult mai eficientă decât funcția *sun* care realizează același lucru relativ la o listă simplu înlanțuită.

Programul de față utilizează o parte din funcțiile apelate de programul BXI6.CPP și anume:

- citcuv;
- incnod;
- elibnod;

și

- cncs.

Funcțiile *adauga* și *sun* se înlocuiesc cu funcțiile *dadauga* și respectiv *dsun*. De asemenea, TNOD se declară ca în exercițiul 11.15.

## PROGRAMUL BX117

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bx11.cpp" /* incnod */
#include "bx12.cpp" /* elibnod */
#include "bx15.cpp" /* dadauga */
#include "bx14.cpp" /* cncs */
#include "bx16.cpp" /* dsun */

TNOD *prim,*ultim;

main() /* citește un text și afișează frecvența cuvintelor din textul
        respectiv */
{
    TNOD *p,*q;

    prim = ultim = 0;
    while((p = dadauga()) != 0)
        if((q = cncs( p -> cuvant )) != ultim ) {
            q -> frecventa++;
            dsun();
        }
}
```



```

}
for( p = prim; p; p = p -> urm)
    printf("cuvintul: %s are frecventa: %d\n",
        p -> cuvant, p -> frecventa);
}

```

11.18 Să se scrie un program care citește cuvintele dintr-un text și scrie numărul de apariție al fiecărui cuvint, în ordinea alfabetică a cuvintelor respective.

Această problemă a fost rezolvată în exercițiul 11.7. În exercițiul respectiv s-a definit funcția *ordlist* care s-a apelat înainte de a afișa frecvența cuvintelor citite. Ea a modificat înălțurile nodurilor listei simplu înălțuite create prin citirea cuvintelor textului, în așa fel încît cuvintele corespunzătoare nodurilor listei să fie ordonate alfabetic.

Programul definit în exercițiul 11.7. diferă de cel definit în exercițiul 11.6. prin prezența funcției *ordlist* și apelul ei înainte de listarea rezultatului.

În exercițiul de față se definește funcția *dordlist* care realizează același lucru ca și funcția *ordlist*, adică modifică înălțurile nodurilor unei liste dublu înălțuite.

Amintim că funcția *ordlist* parcurge lista analizind ordinea cuvintelor corespunzătoare nodurilor vecine. Dacă două cuvinte, care corespund la noduri vecine, nu sînt în ordine alfabetică, atunci se schimbă înălțurile nodurilor respective așa încît ele să fie înălțuite invers în listă.

Inversarea înălțurilor se realizează ca mai jos.

Presupunem că *p* pointează spre nodul curent din listă și *q = p -> urm*, deci *q* pointează spre nodul următor din listă.

Dacă *p -> cuvant* pointează spre un cuvint care este în ordine alfabetică după cuvintul spre care pointează *q -> cuvant*, atunci nodurile *p* și *q* se înălțuiesc invers, deci *p* devine următorul lui *q*. Aceasta implică pașii:

1. Dacă *p -> prec != 0*, atunci *p -> prec -> urm = q*, deoarece următorul precedentului lui *p* devine *q*.
2. Dacă *q -> urm != 0*, atunci *q -> urm -> prec = p*, deoarece precedentul următorului lui *q* devine *p*.
3. *p -> urm = q -> urm*, deoarece următorul lui *q* devine următorul lui *p*.
4. *q -> urm = p*, deoarece *p* devine următorul lui *q*.
5. *q -> prec = p -> prec*, deoarece precedentul lui *p* devine precedentul lui *q*.

6.  $p \rightarrow prec = q$ , deoarece  $q$  devine precedentul lui  $p$ .

## PROGRAMUL BXI18

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bx11.cpp" /* incnod */
#include "bx12.cpp" /* elibnod */
#include "bx15.cpp" /* dadauga */
#include "bx14.cpp" /* cncs */
#include "bx16.cpp" /* dsun */

void dordlist()
/* ordoneaza nodurile listei de tip TNOD in asa fel incit cuvintele
   corespunzatoare nodurilor sa fie in ordine alfabetica */
{
    extern TNOD *prim, *ultim;
    TNOD *p, *q;
    int ind;
    if(prim == 0 )
        return ;
    ind = 1;
    while(ind) {
        ind = 0;
        for(p=prim; p -> urm; ) {
            q = p -> urm; /* q este urmatorul lui p */
            if(strcmp(p -> cuvant, q -> cuvant ) > 0){
                /* se inverseaza inlantuirile nodurilor p si q */
                if(p->prec)
                    p->prec->urm=q;
                if(q->urm)
```

```

        q->urm->prec=p;
    p -> urm = q -> urm;
    q -> urm = p;
    q -> prec = p -> prec;
    p -> prec = q;
    if(p == prim)
        prim = q;
    if(q == ultim)
        ultim = p;
    ind = 1;
}
else /* se trece la nodul urmator deoarece cuvintele sînt in
      ordine alfabetica */
    p = q;
} /* sfîrît for */
} /* sfîrît while */
} /* sfîrît dordlist */

TNOD *prim,*ultim;

main() /* citeşte un text şi afişează frecvenţa cuvintelor diferite în
      ordine alfabetică */
{
    TNOD *p,*q;

    prim = ultim = 0;
    while((p = dadauga()) != 0)
        if((q = cnca(p-> cuvant)) != ultim) {
            q -> frecventa++;
            dsun();
        }
    dordlist();
    for( p = prim; p; p=p -> urm)
        printf("cuvîntul: %-51s are frecvenţa: %d\n",
            p -> cuvant, p -> frecventa);
}

```

## 12. ARBORI

*Arborii*, ca și listele, sînt structuri de date de natură recursivă și dinamică. În multe publicații de specialitate (vezi de exemplu [3]) arborele se definește recursiv.

Prin *arbore* înțelegem o mulțime *finită* și *nevidă* de elemente numite *noduri*:

$$A = \{A_1, A_2, \dots, A_n\}, n > 0$$

care au următoarele proprietăți:

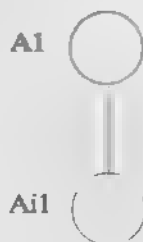
- Există un nod și numai unul care se numește *rădăcina* arborelui.
- Celelalte noduri formează submulțimi disjuncte ale lui  $A$ , care formează fiecare cite un arbore. Arborii respectivi se numesc *sub-arbori* ai rădăcinii.

Un arbore poate fi reprezentat într-un plan. Nodurile se reprezintă prin cercuri. Așa de exemplu, dacă  $A_1$  este rădăcina arborelui  $A$ , atunci figurăm în plan un cerc pe care îl marcăm cu  $A_1$ .

Fie acum:

$$\{A_{i1}, A_{i2}, \dots, A_{ik}\}$$

un subarbore a lui  $A_1$ . Deoarece acesta este el însuși un arbore, fie  $A_{i1}$  rădăcina lui. Lui  $A_{i1}$  îi corespunde un cerc pe care îl marcăm cu  $A_{i1}$ . Acest cerc se figurează sub cerul corespunzător lui  $A_1$ . Cele două cercuri se unesc ca în figura de mai jos.



În mod analog, dacă:

$$\{A_{j1}, A_{j2}, \dots, A_{jr}\}$$

este un alt subarbore al rădăcinii  $A_1$  și  $A_{j1}$  este rădăcina acestui subarbore,

atunci lui  $A_1$  îi corespunde un cerc care se trasează sub cercul corespunzător lui  $A_1$ . De asemenea, cercul marcat cu  $A_1$  se unește și cu cel marcat cu  $A_1$ . De obicei, cercurile marcate cu  $A_1$  și  $A_1$  se află pe aceeași orizontală, ca în figura de mai jos.



Acest proces continuă cu toți subarborii rădăcinii  $A_1$ . Apoi se continuă cu subarborii lui  $A_{i1}$ ,  $A_{j1}$  și așa mai departe.

Într-un arbore există noduri cărora nu le mai corespund subarbori. Un astfel de nod se numește *terminal* sau *frunză*.

În legătură cu arborii s-a încetățenit un limbaj conform căruia un nod *rădăcină* se spune că este un nod *tată*, iar subarborii rădăcinii sînt descendenții acestuia. Rădăcinile descendenților unui nod *tată* sînt *fiu* lui. Astfel, în exemplul de mai sus,  $A_1$  este un nod *tată*, iar  $A_{i1}$  și  $A_{j1}$  sînt *fiu* ai acestuia.

Deci, dacă  $A$  este un nod rădăcină și dacă acesta are  $p$  subarbori a căror rădăcină sînt:

$B_1, B_2, \dots, B_p$

atunci  $A$  este un nod *tată*, iar  $B_1, B_2, \dots, B_p$  sînt *fiu* lui.

Despre nodurile  $B_1, B_2, \dots, B_p$  se spune că sînt *frați*.

Numărul fiilor unui nod *tată* este *ordinul* nodului *tată* respectiv.

În exemplul de mai sus, nodul  $A$  are *ordinul*  $p$ . Un nod *frunză* nu are fii, deci *ordinul* lui este egal cu zero.

O altă noțiune legată de arbori este noțiunea de *nivel*. Rădăcina unui arbore are *nivelul* 1. Dacă un nod are *nivelul*  $n$ , atunci *fiu* lui au *nivelul*  $n+1$ .

Dacă pentru fiecare nod, subarborii săi sînt ordonați (se consideră într-o anumită ordine), atunci arborele se numește *ordonat*. În acest caz se obișnuiește să se spună că rădăcina primului subarbore este *fiul cel mai în vîrstă*, iar rădăcina ultimului subarbore este *fiul cel mai tînăr*.

În multe aplicații practice întâlnim așa numiții *arbori binari*. Un arbore binar este o mulțime finită de elemente care sau este *vidă* sau conține un element numit *rădăcină*, iar celelalte elemente se împart în două submulțimi disjuncte, care fiecare la rândul ei, este un arbore binar. Una din submulțimi este numită *subarboarele stîng* al rădăcinii, iar cealaltă *subarboarele drept*. Arborele binar este ordonat, deoarece în fiecare nod, subarboarele stîng se consideră că precede subarboarele drept. De aici rezultă că un nod al unui arbore binar are cel mult doi fiu și că unul este *fiul stîng*, iar celălalt este *fiul drept*. Fiul stîng este mai în vîrstă decît cel drept. Un nod al unui arbore binar poate să aibă numai un descendent. Acesta poate fi subarboarele stîng sau subarboarele drept.

Cele două posibilități se consideră distincte. Cu alte cuvinte, dacă doi arbori binari diferă numai prin aceea că nodul A dintr-un arbore are ca descendent numai fiul stîng, iar același nod din celălalt arbore are ca descendent numai fiul drept, cei doi arbori se consideră distincți.

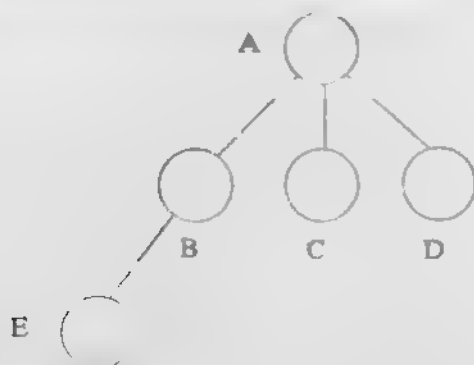
Un arbore binar *nu* se definește ca un caz particular de arbore ordonat. Astfel, un arbore nu este niciodată vid, spre deosebire de un arbore binar care poate fi și vid.

Un arbore ordonat poate fi totdeauna reprezentat printr-un arbore binar (vezi [3]).

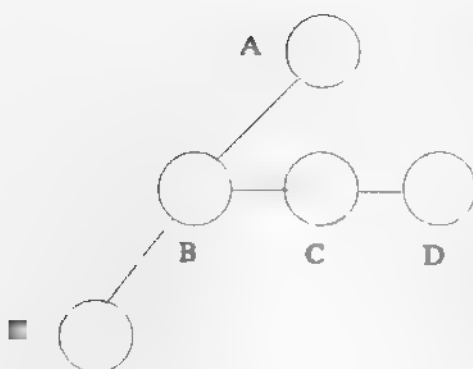
Transformarea se obține destul de simplu și anume:

Se leagă împreună frații descendenți ai unui aceluiași nod tată și se suprimă legăturile lor cu nodul tată, exceptînd legătura primului dintre ei.

De exemplu, arborele din figura de mai jos:



se transformă în arborele binar:



**În arborele transformat:**

**A are pe B ca fiu stîng;**

**B are pe C ca fiu drept și pe E ca fiu stîng;**

**C are pe D ca fiu drept.**

Prin regula de mai sus, rezultă că primul fiu al unui nod tată devine fiul stîng (cel mai vîrstnic). Celelalte legături formează subarbori drepi: al doilea fiu devine fiul drept al primului, al treilea fiu devine fiul drept al celui de al doilea și așa mai departe.

În continuare ne vom ocupa numai de arborii binari.

Un nod al unui arbore binar este o dată structurată de tipul TNOD care se definește în felul următor:

```

typedef struct tnod {
    declarații
    struct tnod *st;
    struct tnod *dr;
} TNOD;
  
```

unde:

- st* - Este pointerul spre fiul stîng al nodului curent.
- dr* - Este pointerul spre fiul drept al aceluiași nod.

Asupra arborilor binari se pot defini mai multe operații dintre care amintim:

- inserarea unui nod frunză într-un arbore binar;
- accesul la un nod al unui arbore;
- parcurgerea unui arbore;

- ștergerea unui arbore.

Operațiile de *inserare* și *accesul* la un nod, au la bază un *criteriu* care să definească locul în arbore al nodului în cauză. Acest criteriu este dependent de problema concretă la care se aplică arborii binari pentru a fi rezolvată. El se definește printr-o funcție pe care o vom numi în continuare funcția *criteriu*. Ea are doi parametri care sînt pointeri spre tipul TNOD. Fie *p1* primul parametru al funcției *criteriu* și *p2* cel de al doilea parametru al ei. Atunci, funcția *criteriu* returnează:

- 1                    - Dacă *p2* pointează spre o dată de tip TNOD care *poate* fi un nod al subarborului stîng al nodului spre care pointează *p1*.
- 1                    - Dacă *p2* pointează spre o dată de tip TNOD care *poate* fi un nod al subarborului drept al nodului spre care pointează *p1*.
- 0                    - Dacă *p2* pointează spre o dată de tip TNOD care *nu* poate fi nod al subarborilor nodului spre care pointează *p1*.

Spre exemplificare, să considerăm că la intrare se află șirul de întregi:

20, 30, 5, 20, 4, 30, 7, 40.

și se pune problema de a determina numărul de apariții al fiecăruia.

Această problemă se poate rezolva dacă se construiește un arbore binar ale cărui noduri au tipul TNOD definit astfel:

```
typedef struct tnod {
    int nr;
    int f;
    struct tnod *st;
    struct tnod *dr;
} TNOD
```

Variabila *nr* are ca valoare unul din numerele citite de la intrare, iar *f* reprezintă frecvența lui de apariție în șirul de la intrare.

Într-un nod frunză *st=dr=0* (pointerul nul).

La început se citește valoarea 20 și arborele va conține un singur nod, corespunzător valorii 20:

```
nr=20
f=1
st=dr=0
```

Figurăm arborele de la această fază astfel:

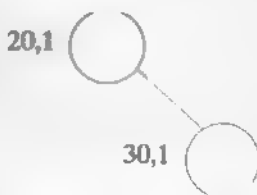




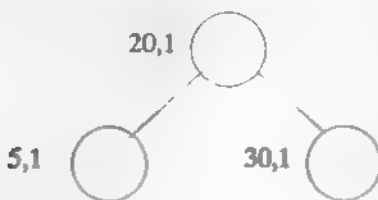
La pasul următor se citește întregul 30 și se construiește data de tip TNOD corespunzătoare lui:

nr=30  
f=1  
st=dr=0

Această dată o vom insera ca fiu drept al rădăcinii arborelui existent de la pasul precedent. Se obține arborele:



Apoi se citește valoarea 5 și nodul corespunzător acesteia se înserează ca fiu stâng al nodului corespunzător valorii 20. Se obține arborele:

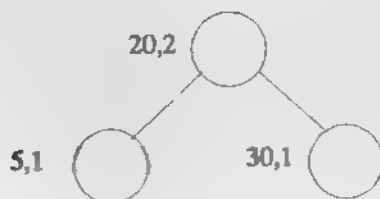


Nodurile se inserează în arbore în așa fel încât dacă un nod corespunde întregului  $n$ , atunci subarborile stâng al lui conține noduri care corespund la valori mai mici decât  $n$ , iar subarborile drept al aceluiași nod conține noduri care corespund la valori mai mari decât  $n$ .

Se observă că arborele de mai sus respectă această regulă. Într-adevăr, rădăcina arborelui corespunde valorii 20, subarborile stâng al ei conține, nodul corespunzător valorii 5 ( $5 < 20$ ), iar subarborile drept al rădăcinii conține nodul corespunzător valorii 30 ( $20 < 30$ ).

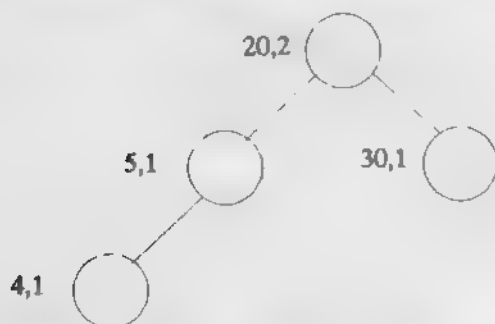
În continuare se citește valoarea 20 și cum există deja în arbore un nod corespunzător acestei valori, se incrementează valoarea lui  $f$  din nodul

respectiv. În felul acesta se obține reprezentarea:



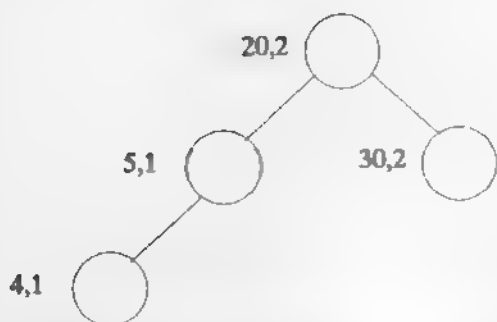
La pasul următor se citește valoarea 4. Se construiește nodul corespunzător valorii 4. Deoarece  $4 < 20$ , nodul curent trebuie inserat în subarborele stâng. Acesta este format din nodul corespunzător valorii 5.

Cum  $4 < 5$ , rezultă că nodul corespunzător lui 4 trebuie inserat în subarborele stâng al nodului corespunzător lui 5. Cum nu există un astfel de subarbor, rezultă că nodul corespunzător lui 4 devine fiu stâng al nodului corespunzător lui 5. Se obține reprezentarea:



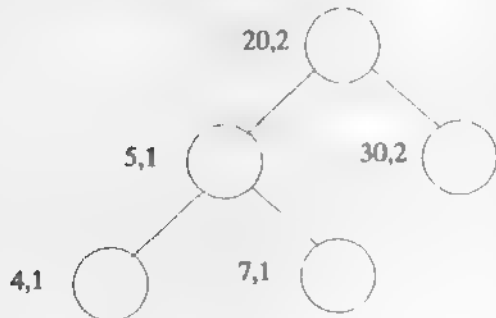
Valoarea următoare citită de la intrare este 30. Ca și în cazurile precedente, se construiește nodul corespunzător valorii citite 30.

Cum  $30 > 20$ , rezultă că nodul construit curent trebuie să se insereze în subarborele drept al nodului corespunzător lui 20. Acest subarbor este format dintr-un singur nod care corespunde lui 30. Întrucât nodul curent corespunde aceleiași valori, rezultă că el nu se mai inserează în arbore ci pur și simplu se incrementează valoarea lui  $f$  pentru nodul corespunzător din arbore. Se obține reprezentarea:



Următorul număr citit este 7. Se construiește nodul corespunzător acestei valori. Deoarece  $7 < 20$ , rezultă că nodul curent se va insera în subarborele stâng al nodului corespunzător lui 20. Rădăcina acestui subarbor este nodul corespunzător lui 5. Cum  $5 < 7$ , rezultă că nodul curent se va insera în subarborele drept al nodului corespunzător lui 5. Întrucât nu există un astfel de subarbor, nodul curent se va insera ca fiu drept al nodului corespunzător lui 5.

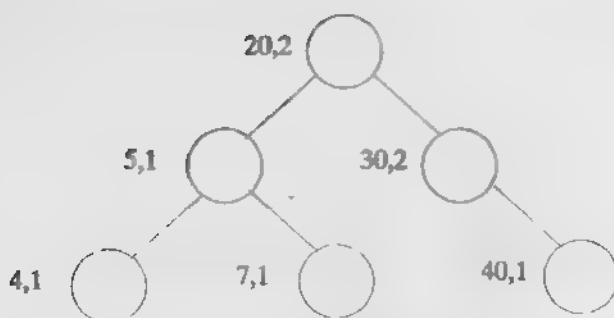
Arborele se reprezintă ca mai jos.



Ultima valoare citită este 40.

Raționând ca mai sus, se ajunge la concluzia că nodul corespunzător valorii 40 se inserează ca fiu drept al nodului corespunzător lui 30.

În final se obține arborele de mai jos.



Se observă că arborele construit respectă regula enunțată mai sus.

Într-adevăr, nodul corespunzător lui 20 ( $n=20$ ) are un subarbore stâng ale cărui noduri corespund valorilor: 5, 4 și 7. Aceste valori sînt toate mai mici decît 20. Același nod, are un subarbore drept ale cărui noduri corespund valorilor: 30 și 40. Ambele sînt mai mari decît 20.

Această proprietate are loc și pentru celelalte noduri care nu sînt noduri frunză. De exemplu, nodul corespunzător valorii 5, are ca subarbore stîng nodul corespunzător lui 4 ( $4 < 5$ ), iar ca subarbore drept nodul corespunzător lui 7 ( $5 < 7$ ).

La construirea arborelui din acest exemplu s-a utilizat următorul criteriu pentru determinarea poziției în care să se insereze în arbore nodul curent (nodul corespunzător valorii curent citite):

- $p1$  este pointer spre un nod al arborelui în care se face inserarea (inițial  $p1$  pointează spre rădăcina arborelui).
- $p2$  este pointer spre nodul curent (nodul de inserat).
- Dacă  $p2 \rightarrow nr < p1 \rightarrow nr$ , atunci se va încerca inserarea nodului curent în subarboarele stîng al nodului spre care pointează  $p1$ .
- Dacă  $p2 \rightarrow nr > p1 \rightarrow nr$ , atunci se va încerca inserarea nodului curent în subarboarele drept al nodului spre care pointează  $p1$ .
- Dacă  $p2 \rightarrow nr = p1 \rightarrow nr$ , atunci nodul curent nu se mai inserează în arbore deoarece există deja un nod corespunzător valorii curent citite.

În acest caz se incrementează  $p1 \rightarrow f$ .

Acest criteriu se realizează imediat printr-o funcție care are ca parametri pointerii  $p1$  și  $p2$  și care returnează valorile:

- 1                      - În cazul descris la punctul c.

- 1 - În cazul descris la punctul d.
- 0 - În cazul descris la punctul e.

Funcția se definește ca mai jos:

```
int criteriu(TNOD *p1, TNOD *p2)
{
    if(p2 -> nr < p1 -> nr)
        return -1;
    if(p2 -> nr > p1 -> nr)
        return 1;
    return 0;
}
```

Din cele de mai sus rezultă că nodul curent nu se mai inserează în arbore, în situația descrisă de punctul e, situație care în general corespunde cazului când funcția *criteriu* returnează valoarea zero. În acest caz nodurile spre care pointează *p1* și *p2* le vom considera *echivalente*.

De obicei, la întâlnirea unei perechi de noduri echivalente, nodul din arbore (spre care pointează *p1*) este supus unei prelucrări, iar nodul curent (spre care pointează *p2*) este eliminat. Pentru a realiza o astfel de prelucrare este necesar să se apeleze o funcție care are ca parametri pointerii *p1* și *p2* și care returnează un pointer spre tipul TNOD (de obicei se returnează valoarea lui *p1*).

Vom numi această funcție *echivalenta*. Ea este dependentă de problema concretă, ca și funcția *criteriu*.

Pentru exemplul de mai sus, funcția *echivalenta* se definește astfel:

```
TNOD *echivalenta(TNOD *p1, TNOD *p2)
{
    p1 -> f++;
    elibnod(p2);
    return p1;
}
```

Funcția *elibnod* este o altă funcție dependentă de problema concretă și care eliberează zonele de memorie ocupate de nodul spre care pointează parametrul ei. Această funcție a fost întâlnită în capitoul precedent și s-a utilizat în același scop pentru nodurile din compunerea listelor.

În sfârșit, o altă funcție dependentă de problema concretă este funcția *incnod*, utilizată și ea în funcțiile din capitoul precedent.

Ea se apelează pentru a încărca datele în nodul curent care urmează a fi inserat în arbore sau pentru a fi prelucrat de funcția *echivalenta* dacă nu are

loc inserarea.

În continuare se definesc funcții pentru operațiile asupra arborilor, amintite mai sus. Toate funcțiile utilizează o variabilă globală care este un pointer spre rădăcina arborelui. Numim *prad* această variabilă. Ea se definește astfel:

**TNOD \*prad;**

În cazul în care într-un program se prelucrează simultan mai mulți arbori, nu se va mai utiliza variabila globală *prad*, interfața dintre funcții realizându-se cu ajutorul unui parametru care este pointer spre tipul TNOD și căruia i se atribuie, la apel, adresa nodului rădăcină al arborelui prelucrat prin funcția apelată.

În paragrafele următoare se definesc funcții care utilizează variabila globală *prad*.

Menționăm că funcțiile *incnod*, *criteriu*, *echivalenta* și *elibnod* pot să aibă și alte denumiri, făcând modificări corespunzătoare în funcțiile care le apelează. De asemenea, aceste funcții pot fi utilizate prin intermediul parametrilor.

Propunem cititorului să realizeze funcțiile din paragrafele acestui capitol în așa fel încât funcțiile *incnod*, *criteriu*, *echivalenta* și *elibnod* să fie apelate prin intermediul parametrilor de tip pointer spre funcție.

## 12.1. Inserarea unui nod frunză într-un arbore binar

Arborele în care se inserează nodul este definit de variabila globală *prad* care are ca valoare adresa de început a zonei de memorie în care se păstrează rădăcina arborelui.

În cazul în care arborele este vid, *prad* are valoarea zero.

Inserarea nodului se realizează conform următorilor pași:

1. Se alocă zonă de memorie pentru nodul care urmează să se insereze în arbore.  
Fie *p* pointerul care are ca valoare adresa de început a zonei respective.
2. Se apelează funcția *incnod*, cu parametrul *p*, pentru a încărca datele curente în zona spre care pointează *p*.

Dacă *incnod* returnează valoarea 1, se trece la pasul 3.

Altfel se revine din funcție cu valoarea zero.

3. Se fac atribuiri:

$p \rightarrow st = p \rightarrow dr = 0$

deoarece nodul de inserat este nod frunză.

4.  $q = \text{prad}$ .

5. Se determină poziția, în arbore, în care să se facă inserarea.

În acest scop se caută nodul care poate fi nod tată pentru nodul curent.

$i = \text{criteriu}(q, p)$ .

6. Dacă  $i < 0$ , se trece la pasul 7.

Altfel se trece la pasul 8.

7. Se încearcă inserarea nodului spre care pointează  $p$  (nodul curent) în subarboarele stîng al nodului spre care pointează  $q$ .

Dacă  $q \rightarrow st$  are valoarea zero, atunci nodul spre care pointează  $q$  nu are subarboare stîng și nodul curent devine fiu stîng al celui spre care pointează  $q$  ( $q \rightarrow st = p$ ).

Se revine din funcție returnîndu-se valoarea lui  $p$ .

Altfel se face atribuirea  $q = q \rightarrow st$  (se trece la fiul stîng al nodului spre care pointează  $q$ ) și se trece la pasul 5.

8. Dacă  $i > 0$ , se trece la pasul 9.

Altfel se trece la pasul 10.

9. Se încearcă inserarea nodului spre care pointează  $p$  (nodul curent) în subarboarele drept al nodului spre care pointează  $q$ .

Dacă  $q \rightarrow dr$  are valoarea zero, atunci nodul spre care pointează  $q$  nu are subarboare drept și nodul curent devine fiu drept al celui spre care pointează  $q$  ( $q \rightarrow dr = p$ ).

Se revine din funcție returnîndu-se valoarea lui  $p$ .

Altfel se face atribuirea  $q = q \rightarrow dr$  (se trece la fiul drept al nodului spre care pointează  $q$ ) și se trece la pasul 5.

10. Nodul curent nu poate fi inserat în arbore.

Se apelează funcția *echivalența* și se revine din funcție cu valoarea returnată de funcția *echivalența*.

Definim mai jos funcția care realizează pașii 1-10.

```

TNOD *insnod()
/* - insereaza un nod in arborele binar spre a carui radacina pointeaza pradi;
- returneaza pointerul spre nodul inserat, pointerul returnat de functia echivalenta sau zero daca nu sint date de incarcat in nod sau la eroare. */
{
    extern TNOD *pradi;
    int i;
    int n;
    TNOD *p,*q;

    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0) &&
        (incnod(p)==1)){ /* 1 */
        p -> st=p -> dr=0;
        if(pradi==0){ /* arbore vid */
            pradi=p; /* nodul curent devine radacina arborelui */
            return p;
        }

/* se determina pozitia in arbore a nodului si se face inserarea daca este cazul */
        q=pradi;
        for(;;){
            if((i=criteriu(q,p))<0)
                if(q -> st==0){ /* se insereaza ca fiu sting */
                    q -> st=p;
                    return p;
                }
            else{ /* se continua cautarea pozitiei in subarborele sting */
                q=q -> st;
                continue;
            }
        }
        if(i>0)
            if(q -> dr==0){ /* se insereaza ca fiu drept */
                q -> dr =p;
                return p;
            }
            else{ /* se continua cautarea pozitiei in subarborele drept */

```



```

        q=q -> dr;
        continue;
    }

/* - nu se face inserarea deoarece nu exista un nod in arbore pentru care
   nodul curent sa poata fi nod fiu (sting sau drept);
   - se apelează funcția echivalenta si se revine din functie. */

        return echivalenta (q,p);
    } /* sfirsit for */
} /* sfirsit if 1 */

if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}

/* eroare la incarcarea datelor in nod sau nu mai sînt date */
elibnod(p);
return 0;

} /* sfirsit insnod */

```

## 12.2. Accesul la un nod al unui arbore binar

Accesul la un nod al unui arbore binar presupune existența unui criteriu care să permită localizarea în arbore a nodului respectiv. De obicei, se poate utiliza funcția *criteriu* folosită la crearea arborelui. Cu ajutorul ei se poate realiza accesul la un nod din arbore echivalent cu un nod dat.

```

TNOD *cauta(TNOD *p)
/* - cauta in arborele spre a carui radacina pointeaza prad, nodul
   echivalent cu cel spre care pointeaza p,
   - returneaza pointerul spre nodul determinat sau zero daca nu exista
   un astfel de nod */
{
    extern TNOD *prad;
    TNOD *q;
    int i;

    if(prad==0)
        return 0; /* arbore vid */
    for(q=prad;q;)

```

```

    if ((i=criteriu(q,p))==0)
        return q;
    else
        if (i<0)
            q=q -> st; /* se cauta in subarborele sting */
        else
            q=q -> dr; /* se cauta in subarborele drept */
    return 0; /* nu s-a gasit in arbore un nod echivalent cu cel spre
        care pointeaza p */
}

```

## 12.3. Parcurgerea unui arbore binar

Prelucrarea informației păstrată în nodurile unui arbore binar se realizează parcurgând nodurile arborelui respectiv. Parcurgerea nodurilor unui arbore binar se poate face în mai multe moduri. Mai jos, indicăm trei moduri de parcurgere a arborilor binari:

- în preordine;
- în inordine;
- în postordine.

*Parcurgerea în preordine* înseamnă accesul la rădăcină și apoi parcurgerea celor doi subarbori ai ei, întâi subarborele sting, apoi cel drept. Subarborii, fiind ei înșiși arbori binari, se parcurg în același mod.

*Parcurgerea în inordine* înseamnă parcurgerea mai întâi a subarborelui sting, apoi accesul la rădăcină și în continuare parcurgerea subarborelui drept. Cei doi subarbori se parcurg în același mod.

*Parcurgerea în postordine* înseamnă parcurgerea mai întâi a subarborelui sting, apoi a subarborelui drept și în final accesul la rădăcina arborelui. Cei doi subarbori se parcurg în același mod.

Accesul la un nod permite prelucrarea informației conținute în nodul respectiv. În acest scop se poate apela o funcție care este dependentă de problema concretă care se rezolvă cu ajutorul parcurgerii arborelui.

Numim *prelucrare* această funcție. Ea este de prototip:

```
void prelucrare(TNOD *p);
```

unde:

*p* - Este pointerul spre nodul a cărui informație se prelucurează.

Un exemplu simplu de funcție *prelucrare* este funcția care afișează informația conținută în nodul spre care pointează *p*.

Reluând exemplul de la începutul capitoului, funcția *prelucrare* poate fi definită ca o funcție care afișează componentele *nr* și *f* ale nodului spre care pointează parametrul ei:

```
void prelucrare(TNOD *p)
{
    printf("numarul=%d aparitii=%d\n",p -> nr,p -> f);
}
```

Folosind această funcție, să exemplificăm cele trei moduri de parcurgere a arborilor binari cu ajutorul arborelui construit la începutul capitoului. Conform celor spuse mai sus, accesul la un nod va însemna apelul funcției *prelucrare*, adică afișarea componentelor *nr* și *f* ale nodului respectiv.

#### **a. Parcurgerea în preordine**

1. Se are acces la rădăcina arborelui, deci se afișează:

numarul=20 aparitii=2

2. Se parcurge subarboarele stîng în preordine.

Aceasta conduce la următorii pași:

- 2.1. Se are acces la rădăcină, adică se afișează:

numarul=5 aparitii=1

- 2.2. Se parcurge subarboarele stîng în preordine.

Aceasta conduce la următorii pași:

- 2.2.1. Se are acces la rădăcină, adică se afișează:

numarul=4 aparitii=1

- 2.2.2. Se parcurge subarboarele stîng.

Acesta este vid de aceea se trece la pasul următor.

- 2.2.3. Se parcurge subarboarele drept.

Acesta este vid (nodul corespunzător lui 4 este frunză) și de aceea se trece la pasul următor.

- 2.3. Se parcurge în preordine subarboarele drept corespunzător lui 5.

Aceasta conduce la următorii pași:

- 2.3.1. Se are acces la rădăcină, adică se afișează:

numarul=7 aparitii=1

2.3.2. Se parcurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

2.3.3. Se parcurge subarborele drept.

Acesta este vid.

Se trece la pasul următor.

3. Se parcurge în preordine subarborele drept corespunzător lui 20.

Aceasta conduce la următorii pași:

3.1. Se are acces la rădăcină, adică se afișează:

numarul=30 aparitii=2

3.2. Se parcurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

3.3. Se parcurge în preordine subarborele drept corespunzător lui 30.

Aceasta conduce la pașii:

3.3.1. Se are acces la rădăcină, adică se afișează:

numarul=40 aparitii=1

3.3.2. Se parcurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

3.3.3. Se parcurge subarborele drept.

Acesta este vid.

Procesul de parcurgere al arborelui se încheie.

Se observă că numerele citite au fost afișate în ordinea:

20, 5, 4, 7, 30, 40.

Introducem următoarele notații:

*Rnr* - Arborele care are ca rădăcină nodul ce corespunde întregului *nr*.

*SRnr* - Subarborele stîng al arborelui *Rnr*.

**b. Parcurgere în înordine**

1. Se parcurge subarborele SR20 în înordine.  
Aceasta conduce la următorii pași:
  - 1.1. Se parcurge subarborele SR5 în înordine.  
Aceasta conduce la următorii pași:
    - 1.1.1. Se parcurge subarborele SR4.  
Acesta este vid.  
Se trece la pasul următor.
    - 1.1.2. Se are acces la R4, adică se afișează:  
numarul=4 aparitii=1
    - 1.1.3. Se parcurge subarborele DR4.  
Acesta este vid.  
Se trece la pasul următor.
  - 1.2. Se are acces la R5, adică se afișează:  
numarul=5 aparitii=1
  - 1.3. Se parcurge subarborele DR5 în înordine.  
Aceasta conduce la următorii pași:
    - 1.3.1. Se parcurge subarborele SR7.  
Acesta este vid.  
Se trece la pasul următor.
    - 1.3.2. Se are acces la R7, adică se afișează:  
numarul=7 aparitii=1
    - 1.3.3. Se parcurge subarborele DR7.  
Acesta este vid.  
Se trece la pasul următor.
2. Se are acces la R20, adică se afișează:  
numarul=20 aparitii=2
3. Se parcurge subarborele DR20 în înordine.

Aceasta conduce la următorii pași:

- 3.1. se parcurge subarborele SR30.

Acesta este vid.

Se trece la pasul următor;

- 3.2. Se are acces la R30, adică se afișează:

numarul=30 aparitii=2

- 3.3. Se parcurge subarborele DR30 în inordine.

Aceasta conduce la următorii pași:

- 3.3.1. Se parcurge subarborele SR40.

Acesta este vid.

Se trece la pasul următor.

- 3.3.2. Se are acces la R40, adică se afișează:

numarul=40 aparitii=1

- 3.3.3. Se parcurge subarborele DR40.

Acesta este vid.

Procesul de parcurgere al arborelui se încheie.

Numerele citite au fost afișate în ordine crescătoare:

4, 5, 7, 20, 30, 40

Acest lucru rezultă imediat din modul în care a fost construit arborele și anume; SRnr se compune din noduri care corespund la numere mai mici decât nr, iar DRnr se compune din noduri care corespund la numere mai mari decât nr. Această proprietate are loc pentru orice nod. Conform definiției parcurgerii *inordine*, nodurile se afișează în ordinea:

- nodurile lui SRnr;
- nodul Rnr;
- nodurile lui DRnr.

Acest principiu se aplică în continuare la subarborii SRnr și DRnr și așa mai departe. În felul acesta se obține o metodă nouă de *sortare*. Ea este mai eficientă decât metoda bulelor în ceea ce privește numărul pașilor.

Sortarea se reduce la construirea arborelui binar și apoi parcurgerea acestuia în inordine. Inversind condițiile de inserare a nodurilor în arbore, se va realiza o sortare în ordine descrescătoare a numerelor citite de la

intrare. Cu alte cuvinte, ordinea sortării se definește cu ajutorul funcției *criteriu*.

La ora actuală există o serie de algoritmi de sortare care sînt mai eficienți decît acesta și de aceea nu mai insistăm asupra lui (vezi[3]).

**c. Parcurgere în postordine**

1. Se parcurge SR20 în postordine.

Aceasta conduce la următorii pași:

1.1. Se parcurge SR5 în postordine.

Aceasta conduce la următorii pași:

1.1.1. Se parcurge SR4.

Acesta este vid.

Se trece la pasul următor.

1.1.2. Se parcurge DR4.

Acesta este vid.

Se trece la pasul următor.

1.1.3. Se are acces la R4, adică se afișează:

numarul=4 aparitii=1

1.2. Se parcurge DR5 în postordine.

Aceasta conduce la următorii pași:

1.2.1. Se parcurge SR7.

Acesta este vid.

1.2.2. Se parcurge DR7.

Acesta este vid.

1.2.3. Se are acces la R7, adică se afișează:

numarul=7 aparitii=1

1.3. Se are acces la R5, adică se afișează:

numarul=5 aparitii=1

2. Se parcurge DR20 în postordine.

Aceasta conduce la următorii pași:

2.1. Se parcurge SR30.

Acesta este vid.

2.2. Se parcurge DR30 în postordine.

Aceasta conduce la următorii pași:

2.2.1. Se parcurge SR40.

Acesta este vid.

2.2.2. Se parcurge DR40.

Acesta este vid.

2.2.3. Se are acces la R40, adică se afișează:

numarul=40 aparitii=1

2.3. Se are acces la R30, adică se afișează:

numarul=30 aparitii=2

3. Se are acces la R20, adică se afișează:

numarul=20 aparitii=2

Procesul de parcurgere a arborelui se încheie.

Numerele citite s-au afișat în ordinea:

4, 7, 5, 40, 30, 20

În următoarele 3 paragrafe se definesc funcții, pentru parcurgerea arborilor în modurile indicate mai sus. Toate cele 3 moduri de parcurgere a arborilor binari au o natură recursivă. Într-adevăr, în oricare din cele 3 moduri se realizează următoarele activități, în fiecare nod al arborelui:

1. Prelucreează nodul N.

2. Parcurgerea în același mod a subarborelui stâng al nodului N.

3. Parcurgerea în același mod a subarborelui drept al nodului N.

Diferența dintre cele 3 moduri constă numai în ordinea în care se realizează activitățile 1-3 de mai sus:

1, 2, 3 - În preordine.

2, 1, 3 - În inordine.

2, 3, 1 - În postordine.

Natura recursivă a acestor moduri de parcurgere decurge din faptul că subarboreii fiecărui nod se parcurg în același mod.

Având în vedere acest fapt, funcțiile de parcurgere a arborilor binari se



definesc cel mai simplu prin funcții recursive.

Fiecare din aceste funcții au ca parametru un pointer spre TNOD:

```
void fparc(TNOD *p);
```

Apelul unei astfel de funcții, atribuie lui *p* adresa de început a zonei de memorie în care se păstrează rădăcina arborelui. Rezultă că funcția *fparc* poate fi apelată printr-o instrucțiune de forma:

```
fparc(prad);
```

### 12.3.1. Parcurgerea arborilor binari în preordine

Numim *preord* funcția care parcurge în preordine un arbore binar. Ea realizează următoarele:

- Dacă pointerul spre rădăcină nu este nul, atunci se execută pașii de mai jos:

1. Se apelează funcția *prelucrare* cu pointerul spre rădăcină.
2. Fiul stâng devine rădăcină și se reapelează funcția *preord* cu pointerul spre noua rădăcină.

În felul acesta se va parcurge în preordine subarboarele stâng.

3. Fiul drept devine rădăcină și se reapelează funcția *preord* cu pointerul spre noua rădăcină.

În felul acesta se va parcurge în preordine subarboarele drept.

- Dacă pointerul spre rădăcină este nul se revine din funcție.

Acești pași se transcriu imediat în limbajul C, ca mai jos:

```
void preord(TNOD *p) /* parcurge arborele binar in preordine */
{
    if(p!=0){
        prelucrare(p); /* prelucreaza radacina */
        preord(p -> st); /* parcurge subarboarele sting in
                           preordine */
        preord(p -> dr); /* parcurge subarboarele drept in
                           preordine */
    }
}
```

### 12.3.2. Parcurgerea arborilor binari în inordine

Numim *inord* funcția care parcurge în inordine un arbore binar. Ea realizează aceeași pași ca și funcția *preord*, dar în altă ordine. Dacă menținem numerotarea pașilor definiți în paragraful precedent, atunci funcția *inord* realizează pașii respectivi în ordinea:

2, 1, 3

În felul acesta se obține funcția de mai jos:

```
void inord(TNOD *p) /* parcurge arborele binar in inordine */
{
    if(p!=0){
        inord(p -> st); /* parcurge subarborele sting in
                        inordine */
        prelucrare(p); /* prelucreaza radacina */
        inord(p -> dr); /* parcurge subarborele drept in
                        inordine */
    }
}
```

### 12.3.3. Parcurgerea arborilor binari în postordine

Numim *postord* funcția care parcurge în postordine un arbore binar. Ea realizează aceeași pași ca și funcțiile *preord* și *inord*, dar în altă ordine și anume:

2, 3, 1

unde prin 1, 2, 3, am notat pașii definiți în paragraful 12.3.

Se obține funcția de mai jos:

```
void postord(TNOD *p) /* parcurge arborele binar
                      in postordine */
{
    if(p!=0){
        postord(p -> st); /* parcurge subarborele sting in
                        postordine */
        postord(p -> dr); /* parcurge subarborele drept in
                        postordine */
        prelucrare(p); /* prelucreaza radacina */
    }
}
```

## 12.4. Ștergerea unui arbore binar

Pentru a șterge un arbore binar este necesară parcurgerea lui și ștergerea fiecărui nod al arborelui respectiv.

Ștergerea unui nod se realizează apelind funcția *elibnod*. Arborele se parcurge în *postordine*. Rezultă că funcția care șterge un arbore binar este asemănătoare cu funcția *postord* definită în paragraful 12.3.3. Deosebirea constă în aceea că, pentru a prelucra rădăcina se apelează funcția *elibnod* în locul funcției *prelucrare*.

```
void stergarb(TNOD *p) /* șterge arborele spre a cărui radacina
                        pointeaza p */
{
    if(p!=0){
        stergarb(p -> st);
        stergarb(p -> dr);
        elibnod(p);
    }
}
```

Menționăm că funcția *stergarb* nu atribuie valoarea zero variabilei globale *prad*. Aceasta este necesar să se realizeze în funcția care apelează funcția *stergarb*.

### Exerciții:

12.1 Se consideră tipul TNOD definit ca mai jos:

```
typedef struct tnod {
    int nr;
    int f;
    struct tnod *st;
    struct tnod *dr;
} TNOD;
```

Să se scrie funcția *incnod* care are antetul:

```
int incnod(TNOD *p);
```

și care realizează următoarele:

- citește un întreg de tip *int* și-l atribuie variabilei *p -> nr*;
- atribuie valoarea 1 variabilei *p->f*;
- returnează:
  - 1 - La întâlnirea sfârșitului de fișier.
  - 1 - Altfel.

Funcția *incnod* apelează funcția *pcit\_int\_lim* definită în exercițiul 8.3.

### FUNCȚIA BXII1

```
int incnod(TNOD *p) /* incarca nodul spre care pointeaza p */
{
    int n;

    if(pcit_int_lim ("nr= ", -32768, 32767,&n) == 0 )
        return -1; /* s-a intilnit EOF */
    p -> nr = n;
    p -> f = 1;
    return 1;
}
```

12.2 Să se scrie funcția *elibnod* care eliberează zona de memorie ocupată de un nod de tipul *TNOD* definit în exercițiul 12.1.

### FUNCȚIA BXII2

```
void elibnod(TNOD *p) /* elibereaza zona de memorie ocupata
                        de nodul spre care pointeaza p */
{
    free(p);
}
```

12.3 Să se scrie funcția *echivalenta* de prototip:

*TNOD \*echivalenta (TNOD \*q,TNOD \*p);*

care realizează următoarele:

- eliberează zona de memorie ocupată de nodul spre care pointează p;
- incrementează valoarea variabilei q -> f;  
(*TNOD* este tipul definit în exercițiul 12.1.);
- returnează valoarea pointerului q.

### FUNCȚIA BXII3

```
TNOD *echivalenta (TNOD *q,TNOD *p)
/* - elibereaza zona de memorie ocupata de nodul spre care pointeaza p,
   incrementeaza valoarea variabilei q -> f;
```

```

- returneaza valoarea lui q. */
{
    elibnod(p);
    q -> f++;
    return q;
}

```

12.4 Să se scrie funcția *prelucrare* care afișează valorile componentelor *nr* și *f* dintr-o dată de tip *TNOD* spre care pointează parametrul ei.

#### **FUNCȚIA BXII4**

```

void prelucrare(TNOD *p) /* afiseaza pe p -> nr si p -> f */
{
    printf("numarul= %d aparitii= %d\n", p -> nr,
           p -> f);
}

```

12.5 Să se scrie funcția *criteriu* definită la începutul acestui capitol pentru datele de tipul *TNOD*.

#### **FUNCȚIA BXII5**

```

int criteriu(TNOD *p1, TNOD *p2)
/* returneaza:
    -1 - daca p2->nr < p1->nr;
    1  - daca p2->nr > p1->nr;
    0  - altfel. */
{
    if(p2 -> nr < p1 -> nr)

/* nodul spre care pointeaza p2 poate fi inserat in subarborele sting al
nodului spre care pointeaza p1 daca subarborele respectiv nu conti-
ne un nod echivalent cu cel spre care pointeaza p2 */
        return -1;

    if(p2 -> nr > p1 -> nr )

/* nodul spre care pointeaza p2 poate fi inserat in subarborele drept al
nodului spre care pointeaza p1 daca subarborele respectiv nu conti-
ne un nod echivalent cu cel spre care pointeaza p2 */
        return 1;
}

```

```

/* nodurile spre care pointeaza p1 si p2 sint echivalente */
return 0;
}

```

12.6 Să se scrie o funcție care inserează un nod într-un arbore binar.

Nodurile arborelui au tipul TNOD.

Această funcție nu depinde de structura nodurilor arborelui. Ea a fost definită în paragraful 12.1.

## FUNCȚIA BX116

```

TNOD *insnod()
/*- insereaza un nod in arborele binar spre a carui radacina pointeaza
   prad;
- returneaza pointerul spre nodul inserat sau pointerul returnat
  de functia echivalenta daca nodul de inserat este echivalent cu unul
  deja aflat in arbore;
- returneaza zero daca nu mai sint date de incarcat sau la eroare. */
{
extern TNOD *prad;
int i;
int n;
TNOD *p,*q;

n = sizeof(TNOD);
if((p=(TNOD *)malloc(n))!=0) &&
    (incnod(p)==1) { /* 1 */
    p -> st = p -> dr = 0;
    if(prad == 0 ) { /* arbore vid */
        prad = p;
        return p;
    }
    q = prad;
    for ( ; ; ) {
        if((i = criteriu(q,p)) < 0)
            if(q -> st == 0 ) {
                q -> st = p;
                return p;
            }
        else {
            q = q -> st;
            continue;
        }
    }
}

```

```

    }
    if( i > 0 )
        if(q -> dr == 0 ) {
            q -> dr = p;
            return p;
        }
        else {
            q = q -> dr;
            continue;
        }
    return echivalenta(q,p);
} /* sfirsit for */
} /* sfirsit if l */
if( p == 0 ) {
    printf("memorie insuficienta\n");
    exit(1);
}
elibnod (p);
return 0;
}

```

**12.7** Să se scrie o funcție care parcurge un arbore binar în preordine.

Această funcție nu depinde de tipul nodurilor arborelui binar și ea a fost definită în paragraful 12.3.1.

## **FUNCȚIA BXII7**

```

void preord(TNOD *p) /* parcurge arborele binar în preordine */
{
    if(p != 0 ) {
        prelucrare(p);
        preord( p -> st);
        preord( p -> dr);
    }
}

```

**12.8** Să se scrie o funcție care parcurge arborele binar în inordine.

Această funcție nu depinde de structura nodurilor arborelui și ea a fost definită în paragraful 12.3.2.

## FUNCȚIA BXII8

```
void inord(TNOD *p) /* parcurge arborele binar in inordine */
{
    if( p != 0 ) {
        inord( p -> st);
        prelucrare(p);
        inord( p -> dr);
    }
}
```

12.9 Să se scrie o funcție care parcurge un arbore binar în postordine.

Această funcție nu depinde de structura nodurilor arborelui și ea a fost definită în paragraful 12.3.3.

## FUNCȚIA BXII9

```
void postord(TNOD *p) /* parcurge arborele binar
                        in postordine */
{
    if( p != 0 ) {
        postord( p -> st);
        postord( p -> dr);
        prelucrare(p);
    }
}
```

12.10 Să se scrie un program care citește un șir de întregi de tip *int* și afișează frecvența de apariție a fiecărui număr citit.

Numerele sînt separate prin caractere albe, iar la sfîrșit se tastează sfîrșitul de fișier.

Rezolvăm această problemă cu ajutorul arborilor binari.

La început se citesc numerele de la intrare și acestea se păstrează în nodurile unui arbore binar de tipul TNOD definit în exercițiul 12.1.

Componenta *nr* are ca valoare numărul citit. La inserarea unui nod în arbore, *f*=1, deoarece nodul nexistind în prealabil în arbore, înseamnă că nici numărul pe care-l conține nodul respectiv n-a mai fost citit înainte.

Dacă la citirea unui număr se constată că acestuia îi corespunde deja un nod în arbore, atunci se incrementează valoarea lui *f* pentru nodul respectiv aflat în arbore.



După construirea arborelui binar, acesta se va parcurge în cele 3 moduri definite în paragraful 12.3., afișându-se frecvența întregilor citiți.

Se va observa că parcurgerea în inordine afișează numerele citite în ordine crescătoare.

## PROGRAMUL BXII10

```
#include <stdio.h>
#include <stdlib.h>

typedef struct tnod {
    int nr;
    int f;
    struct tnod *st;
    struct tnod *dr;
} TNOD;

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bxii1.cpp" /* incnod */
#include "bxii2.cpp" /* elibnod */
#include "bxii3.cpp" /* echivalenta */
#include "bxii4.cpp" /* prelucrare */
#include "bxii5.cpp" /* criteriu */
#include "bxii6.cpp" /* insnod */
#include "bxii7.cpp" /* preord */
#include "bxii8.cpp" /* inord */
#include "bxii9.cpp" /* postord */

TNOD *prad;

main()
/* citește un sir de numere si le pastreaza in nodurile unui arbore binar,
   apoi afiseaza frecventa de aparitie a fiecarui numar citit, parcurgind
   arborele in preordine, inordine si postordine */
{
/* construiește arborele binar */
    prad = 0;
    while( insnod() )
        ;

/* parcurge arborele in preordine */
    printf("\n\n\tpreordine\n\n");
```

```

preord(prad);

/* parcurge arborele in inordine */
printf("\n\n\tinordine\n\n");
inord(prad);

/* parcurge arborele in postordine */
printf("\n\n\tpostordine\n\n");
postord(prad);
}

```

12.11 Se consideră tipul TNOD ca mai jos:

```

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *st;
    struct tnod *dr;
} TNOD

```

Să se scrie funcția *echivalenta* pentru nodurile de tipul TNOD definit mai sus, care realizează următoarele:

Fie antetul:

```
TNOD *echivalenta(TNOD *q, TNOD *p)
```

- funcția eliberează zona de memorie ocupată de nodul spre care pointează *p*;
- incrementează valoarea componentei: *q -> frecventa*;
- returnează valoarea lui *q*.

Această funcție este similară cu funcția definită în exercițiul 12.3.

## FUNCȚIA BXIII1

```

TNOD *echivalenta(TNOD *q, TNOD *p)
/* - elibereaza zona de memorie ocupata de nodul spre care pointeaza p;
   - incrementeaza pe q -> frecventa;
   - returneaza valoarea lui q. */
{
    elibnod (p);
    q -> frecventa++;
    return q;
}

```

### Observație:

Tipul TNOD definit mai sus este similar cu tipul TNOD definit în exercițiul 11.1. Aceste două tipuri diferă numai prin componentele de înlanțuire.

În cazul de față se folosesc pointerii *st* și *dr*, iar în exercițiul 11.1 se utilizează pointerul *urm*.

Celelalte componente (*cuvant* și *frecventa*) sînt identice pentru cele două tipuri de noduri. De aceea, funcția *elibnod* definită în exercițiul 11.2. poate fi utilizată și pentru nodurile de tipul TNOD definit mai sus.

12.12 Să se scrie funcția *prelucrare* care afișează datele care nu sînt de înlanțuire dintr-un nod de tipul TNOD definit în exercițiul 12.11.

### FUNCȚIA BXII12

```
void prelucrare(TNOD *p) /* afiseaza p -> cuvant si
                           p -> frecventa */
{
    static int n = 0;

    printf("cuvintul: %-51s are frecventa: %d\n",
           p -> cuvant, p -> frecventa);
    if(( n+1)%23 == 0 ) {
        printf("pentru a continua actionati o\
              tasta\n");
        getch();
    }
    n++;
}
```

12.13 Să se scrie funcția *criteriu* de antet:

int criteriu(TNOD \*p1,TNOD \*p2)

și care returnează valorile:

-1 - dacă p2 -> cuvant < p1 -> cuvant;

1 - dacă p2 -> cuvant > p1 -> cuvant;

0 - dacă p2 -> cuvant = p1 -> cuvant.

unde.

TNOD

- Este tipul definit în exercițiul 12.11.

## FUNCȚIA BXII13

```
int criteriu(TNOD *p1, TNOD *p2)
/* returneaza:
   -1  -daca p2->cuvant < p1->cuvant;
    1  -daca p2->cuvant > p1->cuvant;
    0  -altfel. */
{
    int i;

    if((i = strcmp(p2 -> cuvant, p1 -> cuvant)) < 0 )
        return -1;
    else
        if( i > 0 )
            return 1;
        else
            return 0;
}
```

12.14 Să se scrie un program care citește cuvintele dintr-un text și afișează numărul de apariții al fiecărui cuvint din textul respectiv.

Cuvintul se definește ca o succesiune de litere mici și/sau mari. Textul se termină prin sfârșitul de fișier

Această problemă a fost rezolvată în capitolul 11, folosind listele simplu înlanțuite și dublu înlanțuite. Programul de față rezolvă această problemă folosind arborii binari.

Rezolvarea cu ajutorul arborilor este similară cu cea folosită în exercițiul 12.10. Diferența constă în aceea că în cazul exercițiului 12.10 se citesc numere de tip *int*, iar în cazul de față se citesc cuvinte.

Programul de față, ca și cel din exercițiul 12.10, apelează o serie de funcții care sînt definite în diverse exerciții. Astfel, funcția *elbnode* este cea definită în exercițiul 11.2. (vezi observația de la exercițiul 12.11.). Funcția *innode*, citește un cuvint și-l păstrează în memoria *heap*, apoi atribuie pointerului *cuvant* adresa zonei de început a memoriei în care se păstrează cuvintul respectiv, iar la componenta *frecventa* i se atribuie valoarea 1. Funcția returnează valoarea -1 la întîlnirea sfîrșitului de fișier și 1 în caz contrar. Această funcție este definită în exercițiul 11.1. Ea apelează funcția *citcuv* definită în exercițiul 10.48.

Celelalte funcții apelate de program sînt definite în capitolul de față.

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *st;
    struct tnod *dr;
} TNOD;

#include "bx48.cpp" /* citcuv */
#include "bx11.cpp" /* incnod */
#include "bx12.cpp" /* elhbnod */
#include "bx1111.cpp" /* echivalenta */
#include "bx1112.cpp" /* prelucrare */
#include "bx1113.cpp" /* criteriu */
#include "bx116.cpp" /* insnod */
#include "bx118.cpp" /* inord */

TNOD *prad;

main()
/* citește cuvintele dintr-un text și le afișează în ordine alfabetică
   împreună cu frecvența de apariție a lor în text */
{
    /* construiește arborele binar */
    prad = 0;
    while(insnod() )
        ;

    /* parcurge arborele în inordine */
    inord(prad);
}

```

#### Observație:

Problema de față se rezolvă mai eficient folosind arborii binari în locul listelor. Aceasta din cauză că operația de căutare a unui cuvânt într-o listă este ineficientă. Într-adevăr, căutarea unui cuvânt într-o listă simplu înălțuită presupune parcurgerea secvențială a nodurilor ei fie pînă la

întilnirea nodului ce conține cuvântul căutat, fie până la sfârșitul listei în cazul în care cuvântul nu este conținut nici într-un nod al listei.

Folosind arborii, procesul de căutare al unui cuvânt în arbore necesită, de obicei, mai puțini pași. În fiecare nod se realizează o ramificare, deoarece se continuă căutarea fie în subarborile stîngi, fie în cel drept, fie se constată că nodul respectiv conține chiar cuvântul căutat.

În felul acesta se observă că la căutarea unui cuvânt o parte din nodurile arborelui rămîn neparcurse.

Utilizarea arborilor devine inefficientă dacă la citirea cuvintelor se obține un arbore *degenerat*. Un arbore binar este degenerat, dacă subarborii lui sînt toți de același fel, de exemplu, toți sînt subarbori stîngi sau toți sînt subarbori drepi. În acest caz, arborele binar este echivalent cu o listă simplu înlănțuită.

În problema de față se obțin arbori degenerați dacă, de exemplu, se citesc cuvintele în ordine alfabetică. Aceasta este puțin probabil cînd se citesc cuvinte dintr-un text normal.

## 13. TABELE

Noțiunea de *tabel* o definim ca o *colecție* de elemente identificabile prin *chei* [3].

Elementele colecției sînt date de *același tip*, care în mod frecvent este un tip utilizator.

Prin *cheie* înțelegem, ca și în cazul listelor, o componentă a elementelor tabelii care are valori *diferite* pentru elemente diferite. În felul acesta, un element din tabel se identifică exact prin valoarea cheii sale.

Elementele unei tabeli se mai numesc și *înregistrări*.

Ca exemple de tabele întâlnite frecvent în diferite aplicații amintim: tabele de cuvinte rezervate, tabele de simboluri, tabele de index pentru fișiere etc.

O problemă importantă care se pune în legătură cu tabelele este aceea a *căutării* unei înregistrări de cheie dată.

*Tabelele de cuvinte rezervate* sînt exemple de tabele *fixe* cu un număr de înregistrări cunoscut dinainte.

Astfel de tabele se întâlnesc în compilatoare și se utilizează pentru a determina dacă un identificator este sau nu un cuvînt rezervat. În acest caz, identificatorul citit din textul de intrare se caută în tabela de cuvinte rezervate pentru a stabili dacă acesta este un cuvînt rezervat sau definit de utilizator. De obicei, procedura de căutare returnează un cod dacă identificatorul căutat s-a aflat în tabela de cuvinte rezervate sau o valoare distinctă de orice cod (de exemplu -1) în cazul în care identificatorul respectiv nu este un cuvînt rezervat.

Tabelele de acest tip se recomandă a fi *ordonate*, deoarece în felul acesta putem aplica metode de căutare eficiente, ca de exemplu *căutarea binară*.

*Tabelele de simboluri* sînt exemple de tabele *dinamice*, care se construiesc pe măsură ce se constată că înregistrarea curentă nu se află în tabelă. Astfel de tabele necesită alte metode de căutare și care în mod frecvent sînt urmate de inserarea în tabelă a înregistrării inexistente.

O tabelă dinamică poate fi organizată ca o listă simplu sau dublu înălțuită, dar în acest caz eficiența procedurii de căutare este foarte mică.

O altă posibilitate este de a organiza tabela inserînd fiecare înregistrare în nodul unui arbore binar. În acest caz, este nevoie de un *criteriu* pe care să-l satisfacă cheile înregistrărilor tabelii respective. Acest criteriu se utilizează

la localizarea în arbore a nodului corespunzător înregistrării curente.

În cazul în care nu există un nod corespunzător înregistrării respective, se inserează în arbore un astfel de nod. În felul acesta, în general, se va mări mult eficiența procedurilor de căutare și inserare a înregistrărilor în arbore. Criteriul care se află la baza căutării într-un arbore binar, de obicei, reduce substanțial numărul nodurilor consultate. Astfel, în fiecare nod are loc una din următoarele 4 posibilități:

- a. Nodul corespunde înregistrării curente.  
Procesul de căutare se încheie, identificându-se nodul căutat.
- b. Căutarea se continuă în subarborele stîng al nodului curent.
- c. Căutarea se continuă în subarborele drept al nodului curent.
- d. Căutarea se întrerupe deoarece nodul curent este un nod frunză (nu există subarbori pentru nodul respectiv).

În acest caz, înregistrării curente nu-i corespunde nici un nod în arbore; înregistrarea curentă, de obicei, se inserează ca fiu stîng sau drept al nodului la care s-a ajuns.

Se observă că în fiecare nod, procesul de căutare poate continua numai în subarborele stîng sau numai în cel drept al nodului respectiv. În felul acesta, la fiecare pas se exclud de la căutare nodurile unui subarbore.

Amintim că se poate ajunge la o eficiență slabă, comparabilă cu cea obținută la utilizarea listelor, dacă arborele degenerază într-o listă (fiecare nod care nu este o frunză are tot timpul numai descendentul stîng sau tot timpul numai descendentul drept).

Degenerarea arborelui se obține atunci cînd înregistrările se inserează în arbore într-o astfel de ordine încît cheile lor satisfac tot timpul criteriul (sau negația acestuia) utilizat la căutarea în arbore. De obicei, această situație are o probabilitate mică de apariție.

O a treia soluție pentru organizarea tabelelor dinamice o constituie așa numitele *tabele de dispersie*. În [3] se descrie, în detaliu, utilizarea tabelelor de acest tip.

În capitolul de față vom prezenta o variantă simplificată de utilizare a tabelelor de dispersie la construirea *tabelor de simboluri*.

## 13.1. Tabela de cuvinte rezervate

O tabelă de cuvinte rezervate este o tabelă ordonată de dimensiune fixă, dinainte cunoscută. Pentru a fixa ideile să presupunem că, cuvintele



rezervate sînt cuvinte împrumutate din limba engleză și sînt utilizate într-un limbaj de programare, avînd fiecare un înțeles predefinit.

În acest caz cheile, precum și înregistrările, se reduc fiecare la cuvintele rezervate. Ordinea impusă cheilor va fi cea *alfabetică*.

Tabela poate fi realizată simplu în limbajul C și anume vom folosi un tablou de pointeri spre caractere și fiecare element al acestuia se inițializează cu adresa de început a zonei de memorie în care se păstrează un cuvînt rezervat. De exemplu, dacă avem în vedere cuvintele rezervate din limbajul C, atunci tabela de cuvinte rezervate se poate declara ca mai jos:

```
char *tcr[]={
    "break",
    "case",
    ...
    "while"
};
```

unde cuvintele rezervate au fost scrise în ordine alfabetică.

Procedura de căutare într-o astfel de tabelă poate fi *căutarea binară* care a fost utilizată pentru un tablou cu elemente numerice în exercițiul 4.42.

În cazul de față se schimbă numai modul de realizare al comparațiilor dintre elementele tabelii și elementul curent care se caută în tabelă. În exercițiul amintit mai sus, comparația se realizează folosind operatorii relaționali obișnuiți ( == și > ) deoarece se compară numere. În cazul de față, pentru a compara două cuvinte rezervate, se utilizează funcția de bibliotecă *strcmp*. Funcția de căutare returnează:

- indicele cuvîntului rezervat dacă acesta a fost găsit în tabelă;
- -1 în caz contrar.

Cu aceste precizări, funcția de căutare în tabela de cuvinte rezervate se poate defini ca mai jos:

```
int cautrez(char *pcrt)
/*- cauta in tabela de cuvinte rezervate cuvintul spre care pointeaza pcrt;
- returneaza indicele cuvintului in tabela sau -1 daca nu exista. */
{
    int inf,sup,i,j;

    static char *tcr[]={
        "break",
        "case",
        ...
        "while"
```

```

};

inf=0;
sup=sizeof(tcr)/sizeof(char *)-1;
for(i=(inf+sup)/2; inf <= sup; i=(inf+sup)/2)
    if((j=strcmp(pcr, tcr[i]))==0)

/* cuvintul cautat s-a gasit in tabela */
    return i;
else
    if(j > 0)

/* cuvintul care se cauta este dupa cel spre care pointeaza tcr[i],
   deci limita inferioara poate fi marita la i+1 */
        inf=i+1;
    else

/* cuvintul care se cauta este inaintea celui spre care pointeaza tcr[i],
   deci limita superioara poate fi micorata la i-1 */
        sup=i-1;

/* cuvintul cautat nu a fost gasit in tabela */
return -1;
}

```

## 13.2. Tabela de dispersie

Tabelele de dispersie au la bază o funcție de transformare a cheilor înregistrărilor:

$hf: K \rightarrow H$

unde:

- $K$  - Este mulțimea cheilor;
- $H$  - Este o mulțime de numere naturale.

În general, funcția  $hf$  nu este injectivă, existând două sau mai multe chei pentru care  $hf$  are aceeași valoare.

Funcția  $hf$  se numește *funcție de dispersie* (în engleză *hashing*), iar utilizarea ei conduce la o metodă numită *tehnică de dispersie* (tehnică de *hashing*).

Prima problemă care se pune în legătură cu tabela de dispersie este aceea

de a alege funcția de dispersie.

În primul rând, se poate presupune că  $0 \leq hf(k) < M$ , pentru orice cheie  $k$ .

Aceasta se obține dacă:

$$hf(k) = f(k) \bmod M,$$

unde:

$f(k)$  - Transformă cheia  $k$  într-un număr natural.

Relația de mai sus definește pe  $hf(k)$ , egal cu restul împărțirii întregi a lui  $f(k)$  prin  $M$ . Pentru a obține o repartizare cât mai uniformă a valorilor funcției  $hf$ , numărul  $M$  se alege să fie prim (vezi [3]).

Despre două chei  $k_i$  și  $k_j$  diferite, se spune că intră în *coliziune*, dacă  $hf(k_i) = hf(k_j)$ .

O funcție de dispersie trebuie să satisfacă următoarele condiții (vezi [3]):

- valoarea ei să se calculeze simplu și rapid;
- să minimizeze numărul coliziunilor.

Numărul coliziunilor este dependent de cheile înregistrărilor, precum și de funcția  $hf$ . Cu cât aceasta realizează o repartuție mai uniformă a valorilor cheilor, cu atât numărul coliziunilor va fi mai mic.

Numărul coliziunilor poate crește dacă numărul  $M$  este prea mic. Pe de altă parte, creșterea exagerată a lui  $M$  conduce la un consum mare de memorie. De aceea,  $M$  trebuie ales printr-un compromis între necesarul de memorie și timpul de căutare suplimentar cheltuit pentru rezolvarea coliziunilor.

Funcția  $f(k)$  se alege în funcție de natura cheilor. Dacă  $k$  este o cheie numerică, atunci se poate alege  $f(k) = k$ .

În cazul în care cheile nu sînt numerice, atunci există mai multe posibilități de a transforma cheile în numere naturale.

Așa de exemplu, se poate considera o codificare numerică a caracterelor unei astfel de chei și prin aceasta fiecare cheie se transformă într-un șir de coduri binare, iar acesta din urmă poate fi interpretat ca reprezentînd un întreg binar fără semn. O astfel de metodă, deși pare simplă, poate conduce la calcule greoaie cu numere mari. De aceea se adoptă metode mai simple. Așa de exemplu, se poate proceda la a păstra numai primele 2-3 coduri binare din reprezentările cheilor sau numai ultimele 2-3 coduri sau un număr mic de coduri din mijlocul cheilor.

O metodă simplă, aplicată frecvent, este aceea de a însuma codurile

caracterelor din compunerea unei chei. Mai jos, vom utiliza și noi acest procedeu simplu pentru a transforma cheile în numere naturale.

### Exemplu:

Fie  $M = 127$ . Presupunem că dorim să construim o tabelă de simboluri. În acest caz cheile înregistrărilor sînt identificatori (succesiuni de litere și eventual și cifre care, de obicei, încep cu o literă). Vom presupune că se utilizează codul ASCII pentru reprezentarea caracterelor. Fie identificatorul AB1. Atunci

$$f(AB1) = 'A' + 'B' + '1' = 65 + 66 + 49 = 180$$

$$hf(AB1) = 180 \% 127 = 53.$$

Evident, identificatorii A1B, BA1, B1A au aceeași dispersie, deoarece:

$$f(AB1) = f(A1B) = f(BA1) = f(B1A) = 180$$

Deci, toți identificatorii de mai sus intră în coliziune.

Problemele implicate de coliziune pot fi rezolvate în mai multe moduri (vezi [3]).

În cartea de față adoptăm o soluție simplă și anume aceea de a rezolva coliziunile cu ajutorul listelor simplu înlanțuite. Astfel, toate înregistrările pentru care cheile au aceeași dispersie se înserează într-o listă simplu înlanțuită.

Vom avea deci, mai multe liste simplu înlanțuite, fiecare listă conținând înregistrările ale căror chei au aceeași dispersie.

Pentru a putea gestiona înregistrările din aceste liste este necesar să cunoaștem pointerul spre primul nod al fiecărei astfel de liste.

Numărul maxim al acestor liste este  $M$ , deoarece funcția  $hf$  are  $M$  valori:  $0, 1, \dots, M-1$ . Pointerii spre începuturile listelor respective se păstrează într-un tablou de  $M$  elemente. Numim *thash* acest tablou. Atunci  $thash[i]$  are ca valoare pointerul spre începutul listei ce conține înregistrările pentru care cheile au dispersia egală cu  $i$ .

Reluînd exemplul de mai sus,  $thash[53]$  are ca valoare pointerul spre începutul listei care conține înregistrările cu cheile: AB1, BA1 și B1A.

Inițial, aceste liste nu există și în consecință elementele  $thash[i]$  pentru  $i=0, 1, 2, \dots, M-1$  au valoarea zero.

În capitolul 11 s-au utilizat variabilele globale *prim* și *ultim* pentru a defini capetele unei liste simplu înlanțuite. În cazul de față un element al tabloului *thash* joacă același rol ca și variabila *prim*. Date fiind operațiile avute în vedere asupra listelor care conțin înregistrările aflate în coliziune,

se constată că nu avem nevoie de variabila *ultim*.

Procedura de căutare într-o tabelă de dispersie se realizează în următorii pași:

1. Dacă  $thash[h]=0$ , unde  $h$  este dispersia cheii curente, atunci înregistrarea respectivă nu există în tabelă și funcția de căutare returnează valoarea zero.
2. Dacă  $thash[h]$  este diferit de zero, atunci acesta este pointerul spre începutul listei în care se află înregistrările pentru care cheile au dispersia egală cu  $h$ .

Căutarea înregistrării curente se face în această listă începând cu nodul spre care pointează  $thash[h]$ .

Această căutare se realizează secvențial, nod după nod, până când se găsește o înregistrare cu aceeași cheie ca și cea curentă sau până când se ajunge la ultimul nod al listei, lucru care se întâmplă când înregistrarea nu este prezentă în lista respectivă.

În primul caz funcția de căutare returnează pointerul spre înregistrarea găsită, iar în al doilea caz returnează valoarea zero.

Din cele de mai sus rezultă că procedura de căutare este cu atât mai eficientă, cu cât listele cu înregistrările aflate în coliziune sînt mai mici. Aceasta este funcție de constanta  $M$  și de funcțiile  $hf$  și  $f$  definite mai sus.

De obicei, în cazul tabelelor de simboluri utilizate în compilatoare,  $M$  se alege sub 1000. De asemenea, însumarea codurilor caracterelor identificatorilor s-a dovedit a fi o soluție simplă și eficientă pentru calculul dispersiei. În acest caz funcția  $hf$  de calcul a dispersiei se definește ca mai jos:

```
#define M 127
unsigned hf(char *s)
{
    unsigned h;
    for(h = 0; *s;)
        h += *s++;
    return h%M;
}
```

În continuare, definim funcția de căutare în tabela de dispersie, presupunind că aceste chei sînt șiruri de caractere.

Numim *hcaut* această funcție. Ea returnează pointerul spre înregistrarea căutată sau zero dacă înregistrarea respectivă nu există în tabelă.

O înregistrare are tipul **TNOD** definit ca în cazul listelor simplu

înlanțuite:

```
typedef struct tnod {  
    declaratii  
    char *cheie;  
    declaratii  
    struct tnod *urm;  
} TNOD;
```

Tabloul *thash* are *M* elemente și elementele lui sînt pointeri spre tipul *TNOD*. Vom presupune că acest tablou este global.

Inițial elementele tabloului *thash* au valoarea zero (toate listele sînt vide). Acest tablou se definește astfel:

```
TNOD *thash[M];
```

Funcția *hcaut* are ca parametri pointerul spre șirul de caractere din compunerea cheii (se caută înregistrarea a cărei cheie coincide cu acest șir de caractere) și valoarea dispersiei acestui șir.

Cu aceste precizări, funcția de căutare se definește astfel:

```
TNOD *hcaut(char *s, unsigned h)  
/* - cauta inregistrarea pentru care cheie pointeaza spre un sir de  
   caractere identic cu cel spre care pointeaza s;  
   - returneaza pointerul spre inregistrarea respectiva sau zero daca  
   nu exista o astfel de inregistrare. */  
{  
    TNOD *p;  
  
    /* cauta inregistrarea in lista spre care pointeaza thash[h] */  
    for(p = thash[h]; p; p = p -> urm)  
        if(strcmp(s, p -> cheie) == 0)  
  
            /* s-a gasit inregistrarea cautata */  
            return p;  
  
    /* nu exista o inregistrare pentru care cheie sa poarte spre un sir  
       identic cu cel spre care pointeaza s */  
    return 0;  
}
```

Operația de introducere a unei înregistrări într-o tabelă de simboluri se va realiza printr-o funcție pe care o numim *intab*. Ca și în cazul listelor, funcția *intab* va utiliza funcțiile *incnod* și *elibnod*, prima pentru a încărca datele înregistrării în zona de memorie alocată prin funcția *malloc*, iar a

doua pentru a elibera o astfel de zonă.

Deoarece tabloul *thash* conține pointeri spre începutul listelor și nu spre sfârșitul lor, va fi util ca operația de inserare a nodului nou în listă să se facă înaintea primului ei nod, adică folosind o funcție analogă cu funcția *inprim* definită în paragraful 11.1.3.1.

Astfel, dacă *p* pointează spre nodul care se inserează în listă, iar *h* este dispersia cheii lui, atunci știm că *thash[h]* pointează spre primul nod al listei în care urmează să se insereze înregistrarea spre care pointează *p*. Cum înregistrarea se inserează înaintea primului nod al listei, va trebui ca după inserare *p* -> urm să poarte spre acest prim nod, deci se face atribuirea:

*p* -> urm = *thash[h]*.

Totodată, înregistrarea care se inserează devine primul nod al aceleiași liste, deci în continuare va trebui ca:

*thash[h]* = *p*.

În felul acesta, operația de inserare a înregistrării curente se realizează conform pașilor de mai jos:

1. Rezervă zonă pentru înregistrarea curentă folosind funcția *malloc*.
2. Încarcă datele în zona rezervată la punctul 1.
3. Caută înregistrarea curentă în tabelă.

Dacă există, atunci se revine din funcție returnându-se pointerul spre înregistrarea găsită.

Altfel se inserează înregistrarea curentă în tabelă și se returnează pointerul spre ea.

Funcția returnează valoarea zero dacă nu mai există înregistrări sau s-a depistat o eroare la încărcarea datelor în înregistrarea curentă.

Definim mai jos funcția *intab*.

```
TNOD *intab()
```

```
/* - determina înregistrarea curentă și o inserează dacă nu a fost găsită  
   în tabelă;
```

```
- returnează pointerul spre înregistrarea respectivă sau NULL în  
   cazul când nu mai există înregistrări sau la eroare. */
```

```
TNOD *p,*q;
```

```
unsigned h;
```

```
/* rezervă zona și încarcă datele înregistrării curente */
```

```
if(((p=(TNOD *)malloc(sizeof(TNOD)))!=0)
```

```

        &&(incnod(p)==1)){

/* calculeaza dispersia cheii */
    h = hf(p -> cheie);

/* - daca thash[h]!=0, atunci exista in tabela inregistrari a caror chei
    au dispersia h;
    - in acest caz se cauta inregistrarea in tabela. */
    if(thash[h])
        if((q=hcaut(p -> cheie,h))!=0)

/* inregistrarea exista in tabela */
        return echivalenta(q,p);
    else{

/* - inregistrarea nu exista in tabela;
    - se insereaza inaintea nodului spre care pointeaza thash[h]. */
        p -> urm = thash[h];
        thash[h] = p;
        return p;
    }
    else{

/* nu exista inregistrari care sa intre in coliziune cu inregistrarea
    curenta */
        thash[h] = p;
        p -> urm = 0;
        return p;
    }
}
if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
}

/* eroare la incarcarea datelor sau nu mai sînt inregistrari */
elibnod(p);
return 0;
}

```

Funcția *echivalenta* se apelează în cazul în care în tabelă există o înregistrare de aceeași cheie cu cea a înregistrării curente. În general, această funcție are ca efect modificarea sau actualizarea datelor înregistrării din tabelă cu cele din înregistrarea curentă.



Ea a fost utilizată și în cazul inserării nodurilor în arbori binari (vezi capitolul 12).

În cazul în care înregistrarea curentă nu conține date pentru modificarea înregistrării din tabelă care-i corespunde, funcția *echivalenta* se reduce la ștergerea înregistrării curente, ca mai jos:

```
TNOD *echivalenta(TNOD *q, TNOD *p)
/* sterge înregistrarea spre care pointeaza p și returnează
   valoarea lui q */
{
    elibnod(p);
    return q;
}
```

#### Observații:

1. Sistemul format din listele care conțin înregistrările aflate în coliziune și din tabloul de pointeri spre primele înregistrări ale acestor liste, constituie o *tabelă de dispersie*.

La baza inserării și accesului la o înregistrare din tabela de dispersie se află funcția *hf* de calcul a dispersiei.

2. Tabela de dispersie poate fi organizată înlocuind listele cu înregistrările aflate în coliziune prin arbori binari.

În acest caz, elementele tabloului *thash* au ca valori pointeri spre rădăcinile acestor arbori.

În felul acesta operațiile de căutare sînt, de obicei, mai performante (excepție fac cazurile cînd arborii sînt degenerați, dar această posibilitate este extrem de redusă).

În felul acesta, un arbore mare s-a descompus în mai mulți (cel mult *M*) care sînt relativ mici. Astfel, căutarea unei înregistrări nu se mai face într-un arbore mare ci într-unul mult mai mic care a fost selectat pe baza unui calcul simplu.

#### Exerciții

- 13.1 Să se scrie o funcție care calculează dispersia unui șir de caractere prin însumarea codurilor caracterelor șirului respectiv.

Accasta este funcția *hf* definită în paragraful 13.2.

#### FUNCȚIA BXIII1

```
unsigned hf(char *s) /* calculează dispersia șirului spre care
```

```

                                pointeaza s */
{
    unsigned h;

    for(h=0; *s;)
        h += *s++;
    return h%M;
}

```

#### Observație:

*M* este o constantă simbolică și reprezintă numărul de elemente al tabloului de pointeri *thash*.

13.2 Să se scrie funcția *hcaut* care caută o înregistrare de cheie dată într-o tabelă de dispersie.

Funcția de față a fost definită în paragraful 13.2.

#### FUNCȚIA BXIII2

```

TNOD *hcaut(char *s,unsigned h)
/* - cauta inregistrarea de cheie identica cu sirul spre care pointeaza s
   si a carei dispersie este h;
   - returneaza pointerul spre inregistrarea respectiva sau zero
   daca inregistrarea este absenta. */
{
    TNOD *p;

    /* cauta inregistrarea in lista spre care pointeaza thash[h] */
    for(p=thash[h];p;p=p->urm)
        if(strcmp(s,p->cuvant)==0)

    /* s-a gasit inregistrarea curenta */
        return p;

    /* nu exista inregistrarea cautata */
    return 0;
}

```

13.3 Să se scrie funcția *intab* care înserează o înregistrare într-o tabelă de dispersie.

Funcția a fost definită în paragraful 13.2.

## FUNCȚIA BXIII3

```
TNOD *intab()  
/* - determina inregistrarea curenta si o insereaza daca nu exista in tabela  
   de dispersie;  
   - returneaza pointerul spre inregistrarea respectiva sau zero in  
   cazul cind nu mai exista inregistrari sau la eroare. */  
{  
    TNOD *p,*q;  
    unsigned h;  
  
    /* rezerva zona si incarca datele inregistrarii */  
    if((p=(TNOD *)malloc(sizeof(TNOD))) &&  
        (incnod(p)==1)){  
  
        /* calculeaza dispersia cheii inregistrarii curente */  
        h=hf(p -> cuvant);  
  
        /* daca thash[h]!=0, se cauta inregistrarea in tabela */  
        if(thash[h])  
            if(q=hcaut(p -> cuvant,h))  
  
        /* s-a gasit inregistrarea cautata */  
            return echivalenta(q,p);  
        else{  
  
        /* - inregistrarea curenta nu exista in tabela;  
           - se insereaza */  
            p -> urm=thash[h];  
            thash[h]=p;  
            return p;  
        }  
        else{  
  
        /* nu exista inregistrari care sa intre in coliziune cu inregistrarea  
           curenta */  
            p -> urm =0;  
            thash[h]=p;  
            return p;  
        }  
    }  
    if(p==0){  
        printf("memorie insuficienta\n");  
        exit(1);  
    }
```

```

    }
    elibnod(p);
    return 0;
}

```

13.4 Să se scrie un program care citește cuvintele unui text și scrie numărul de apariții al fiecăruia dintre ele.

Această problemă a fost rezolvată în capitolele precedente folosind:

- listele simplu înlănțuite;
- listele dublu înlănțuite;
- arborii binari.

În exercițiul de față vom folosi o tabelă de dispersie. Problema, ca și în cazul utilizării celorlalte metode, se rezolvă conform pașilor de mai jos:

1. Se citește cuvântul curent;
2. Se caută cuvântul curent în tabela de dispersie.

Dacă el există deja în tabelă, atunci se incrementează numărătorul său.

Altfel se înregistrează în tabela corespunzătoare cuvântului curent citit și se inițializează numărătorul lui cu valoarea 1.

Acești pași se execută până la întâlnirea sfârșitului de fișier care marchează sfârșitul textului.

În programul de față se folosesc unele funcții care s-au utilizat și în programele precedente. Aceste funcții sint:

*citcuv, incnod, elibnod și echivalenta.*

#### PROGRAMUL 1031

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *urm;
} TNOD;

```

```

#define M 127

TNOD *thash[M];

#include "BX40.CPP" /* citcuv */
#include "BXI1.CPP" /* incnod */
#include "BXI2.CPP" /* ehbnod */
#include "BXIII1.CPP" /* echivalenta */
#include "BXIII1.CPP" /* hf */
#include "BXIII2.CPP" /* hcaut */
#include "BXIII3.CPP" /* intab */

main() /* citește un text și afișează frecvența cuvintelor citite */
{
    TNOD *p;
    int i,j;

    /* inițializează tabloul thash cu pointerul nul */
    for(i=0;i < M; i++)
        thash[i]=0;

    /* - creează tabela de dispersie;
       - fiecărui cuvânt citit din textul sursă îi corespunde o înregistrare
       de tipul TNOD. */
    while(intab())
        ;

    /* se listează cuvintele citite și frecvența lor */
    j=0;
    for(i=0;i < M;i++)
        for(p=thash[i];p;j++,p=p->urm){
            printf("cuvântul: %-51s are\
                    frecvența=%d\n", p->cuvant,
                    p->frecvența);
            if((j+1)%23==0){
                printf("Acționați o tastă pentru a\
                        continua\n");
                getch();
            }
        }
}

```

13.5 Să se scrie un program care citește cuvintele dintr-un text și afișează

următoarele date despre fiecare cuvânt distinct citit:

- caracterele cuvântului;
- dacă este rezervat sau nu;
- numărul liniei în care apare prima dată cuvântul.
- numărul coloanei în care începe cuvântul în linia afișată;
- numărul de apariții al cuvântului în textul respectiv.

Cuvintele care se citesc se vor păstra într-o tabelă de dispersie. Înregistrările acestei table sînt de tipul TNOD definit mai jos:

```
typedef struct tnod {  
    char *cuvant;  
    int frecventa;  
    int nr_linie;  
    int nr_col;  
    Boolean tip;  
} TNOD;
```

Variabila *tip* are valoarea 1 dacă cuvântul este rezervat și 0 în caz contrar.

În program se definește o tabelă de cuvinte rezervate. Acestea sînt alese din setul de cuvinte rezervate ale limbajului C.

Programul se realizează în conformitate cu pașii de mai jos:

1. Se citește un cuvînt de la intrare.  
Dacă s-a întîlnit sfîrșitul de fișier se trece la pasul 5.  
Altfel se continuă cu pasul 2.
2. Se construiește înregistrarea de tip TNOD pentru cuvîntul citit corect.
3. Se caută înregistrarea curentă în tabela de dispersie.  
Dacă există, se incrementează frecvența cuvîntului din înregistrarea găsită în tabela de dispersie.  
Altfel se inserează înregistrarea în tabelă.
4. Se trece la pasul 1.
5. Se listează datele păstrate în tabela de dispersie.

Prin cuvînt se înțelege o succesiune de litere mici și/sau mari.

```
#include <stdio.h>
```

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>

typedef enum {False,True} Boolean;

typedef struct tnod{
    char *cuvant;
    int frecventa;
    int nr_lin;
    int nr_col;
    Boolean tip;
    struct tnod *urm;
} TNOD;

#define M 127

TNOD *thash[M];
int linie,coloana;

int incnod(TNOD *p) /* citeste cuvintul curent si construieste
                    inregistrarea corespunzatoare lui */
{
    int c,i,j;
    char t[255];
    char *q;
    int inf,sup;
    static char *tcr[]={
        "break","case","char","continue","default",
        "do","double", "else","extern","float","for",
        "goto","if","int","long","register","return",
        "short","sizeof","static","struct","switch",
        "typedef","union","unsigned","void","while"
    };

    inf=0;
    sup=sizeof(tcr)/sizeof(char *)-1;
    p -> cuvant=0;
    while((c=getchar())!=EOF){
        coloana++;
        if(c >= 'A' && c <= 'Z' || c >= 'a' &&
           c <= 'z')

```

```

        break;
    if(c=='\n'){
        linie++;
        coloana=0;
    }
}
if(c==EOF)
    return -1;
p -> nr_lin = linie;
p -> nr_col = coloana;
for(i=0; i < 255; i++){
    if((c >= 'A' && c <= 'Z') && !(c >= 'a' &&
        c <= 'z')){
        if(c=='\n'){
            linie++;
            coloana=0;
        }
        break;
    }
    t[i]=c;
    c=getchar();
    coloana++;
}
if(c==EOF)
    return -1;
t[i++]='\0';
if((q=(char *)malloc(i))==0){
    printf("memorie insuficienta\n");
    exit(1);
}
strcpy(q,t);
p -> cuvant=q;
p -> frecventa=1;

/* cauta cuvintul citit in tabela de cuvinte rezervate */
for(i=(inf+sup)/2; inf <= sup; i=(inf+sup)/2){
    if((j=strcmp(t, tcr[i]))==0)
        break;
    if(j < 0)
        sup=i-1;
    else
        inf=i+1;
}

```



```

    if(j)

/* nu este cuvint rezervat */
    p -> tip = False;
else /* cuvint rezervat */
    p -> tip = True;
    return 1;
} /* sfirsit incnod */

#include "BXI2.CPP" /* elibnod */
#include "BXIII1.CPP" /* echivalenta */
#include "BXIII1.CPP" /* hf */
#include "BXIII2.CPP" /* hcaut */
#include "BXIII3.CPP" /* intab */

main() /* - citeste cuvintele unui text;
        - afiseaza pentru fiecare cuvint distinct din text:
            - caracterele cuvintului;
            - daca este rezervat sau nu;
            - numarul liniei si coloanei in care apare cuvintul
              de prima data;
            - numarul de aparitii al cuvintului respectiv in textul
              citit. */
{
    int i,j;
    TNOD *p;

/* initializari */
    linie = 1;
    coloana = 0;
    for(i = 0; i < M; i++)
        thash[i] = 0;

/* se citesc cuvintele de la intrare si se construiesc tabela de dispersie */
    while(intab())
        ;

/* afiseaza datele din tabela de dispersie */
    j = 0;
    for(i = 0; i < M; i++)
        for(p = thash[i]; p; j++, p = p -> urm){
            printf("\t%s\n", p -> cuvint);
            if(p -> tip == True)

```

```

        printf(" rezervat");
    else
        printf("nerezervat");
    printf(" linia:%6d\t coloana=%6d\t\
        aparitii:%d\n", p -> nr_lin,
        p -> nr_col, p -> frecventa);
    if((j+1)%10==0){
        printf("Actionati o tasta pentru a\
            continua\n");
        getch();
    }
}
} /* sfirsit main */

```

## 14. SORTARE

În prezent există mai multe metode de sortare a datelor. Prin *sortare* înțelegem aranjarea datelor într-o anumită ordine. În acest scop considerăm că datele sînt o colecție de elemente de un anumit tip și fiecare element conține o dată sau chiar mai multe, în raport cu care se realizează ordonarea. O astfel de dată se numește *cheie*.

Sortarea se poate realiza:

1. Fie aranjînd datele care se sortează în așa fel încît cheile lor să corespundă ordinii dorite.
2. Fie ordonînd un tablou de pointeri spre datele care trebuiesc sortate în așa fel încît considerîndu-i pe aceștia în ordinea crescătoare a indicilor tabloului, datele spre care pointează să formeze o mulțime ordonată în conformitate cu ordinea dorită.

Un exemplu de sortare, de felul celui indicat la punctul 1 de mai sus, este exercițiul 4.40. În acest exemplu se definește o funcție care ordonează crescător elementele unui tablou de tip *double*.

Exercițiul 8.14. este un exemplu de sortare folosînd un tablou de pointeri așa cum se indică la punctul 2 de mai sus. În acest caz se sortează un șir de cuvinte ordonînd pointerii care pointează spre ele. Aceștia sînt păstrați într-un tablou de pointeri spre caractere. În timpul sortării se fac permutări ale acestor pointeri în așa fel încît, în final, acești pointeri dacă sînt considerați în ordinea crescătoare a indicilor, atunci cuvintele spre care pointează se află în ordine alfabetică.

În ambele cazuri metoda de sortare a fost aceeași, metoda bulelor. De aceea, cele două cazuri de mai sus nu trebuiesc considerate ca fiind metode de sortare diferite.

Metoda bulelor are la bază compararea a două chei învecinate (de indici succesivi sau din noduri succesive). Dacă acestea nu sînt în ordinea cerută, atunci se permută elementele respective sau pointerii spre ele. O caracteristică a acestui algoritm este faptul că la fiecare parcurgere a șirului de date care se sortează, cel puțin un element ajunge să-și ocupe poziția sa finală în conformitate cu ordonarea cerută. În cazul exercițiilor amintite mai sus nu s-a ținut seama de această proprietate. Propunem cititorului să rescrie exercițiile respective ținînd seama de faptul că la fiecare parcurgere cel puțin un element bine determinat este poziționat pe locul lui final. Aceasta înseamnă că la fiecare parcurgere se poate diminua cu 1 numărul elementelor de analizat în următoarea parcurgere. Aceasta conduce la o

îmbunătățire a eficienței algoritmului de sortare prin metoda bulelor. Cu toate acestea, această metodă rămâne printre cele cu cel mai mare număr de permutări.

O altă metodă de sortare este legată de folosirea *arborilor binari*. Datele care se sortează se inserează în nodurile unui arbore binar și apoi acesta se listează în *inordine*. O astfel de metodă a fost utilizată în exercițiul 12.14. Programul definit în acest exercițiu citește cuvintele dintr-un text și apoi le afișează în ordine alfabetică.

În general, sortarea cu ajutorul *arborilor binari* este mai eficientă decât metoda bulelor.

Există diverși algoritmi de sortare bazați pe arbori. Pentru detalii cititorul poate consulta [3].

În capitolul de față prezentăm două metode de sortare utilizate frecvent în diferite aplicații. Aceste metode sînt relativ simple și în același timp asigură o eficiență mai bună decât metoda bulelor.

Una dintre aceste metode este cunoscută sub denumirea de *sortare Shell* sau *sortare cu micșorarea incrementului*.

Cealaltă metodă se numește *quick sort* (sortare rapidă).

În continuare, vom avea în vedere numai sortări cu chei numerice întregi. Aceasta nu este o restricție esențială, deoarece sortările cu chei nenumerice se realizează similar, folosind tablouri de pointeri.

## 14.1. Sortare Shell

Metoda *Shell* a fost publicată de către Donald L. Shell, în lucrarea [19]. Metoda pornește de la analiza critică a metodei bulelor. În metoda bulelor, se compară elementele vecine și dacă nu sînt în ordinea cerută, atunci ele se permută. De aceea, elementele care nu sînt în ordine corectă se deplasează cu o singură poziție. Pentru a îmbunătăți acest procedeu, ar trebui să realizăm deplasări cu mai multe locuri ale elementelor, la o permutare a lor. Acest lucru se poate realiza dacă în loc de a compara elemente învecinate, comparăm elemente aflate la o anumită distanță între ele. În cazul în care ele nu sînt în ordinea cerută, ele se vor permuta. În felul acesta elementele se deplasează făcînd salturi mai mari decât o poziție.

Distanța dintre elementele comparate se numește *increment*. Incrementul se micșorează după o parcurgere a șirului de elemente care se sortează și se reia parcurgerea de la începutul șirului. De aici rezultă denumirea *sortare cu micșorarea incrementului*.

Înainte de a defini exact metoda de sortare a lui Shell, vom considera un exemplu.

Fie de sortat, în ordine crescătoare, șirul de numere:

173   25   4   300   256   83   95   14   415   3

La început trebuie ales incrementul.

O metodă simplă, dar nu chiar cea mai eficientă, este de a fixa valoarea acestuia la jumătate din numărul total al elementelor de sortat. În cazul de față incrementul va fi:

$inc = 5$

Apoi, incrementul se înjumătățește după fiecare parcurgere a șirului. Algoritmul se termină cind  $inc = 0$ .

Algoritmul începe cu a compara primul element al șirului (173) cu al 6-lea (incrementul fiind 5) (83). Cum  $173 > 83$ , se face permutarea elementelor respective:

83   25   4   300   256   173   95   14   415   3

Compararea următoare se realizează pentru elementul al 2-lea cu al 7-lea, deci se compară 25 cu 95. Ele fiind în ordinea cerută, nu se permută.

Apoi se compară elementele următoare, adică 4 cu 14. Ele fiind în ordinea cerută nu se permută. În continuare se compară elementele următoare, adică 300 cu 415, care și ele sînt în ordinea cerută și deci nu se permută. Elementele următoare 256 și 3 nu sînt în ordine crescătoare și în consecință ele se permută. Se obține:

83   25   4   300   3   173   95   14   415   256

Se înjumătățește incrementul și se obține:

$inc = 2$

Se reia parcurgerea șirului ca mai jos.

Se compară primul element (83) cu al treilea (4). Deoarece  $83 > 4$ , se permută elementele respective:

4   25   83   300   3   173   95   14   415   256

Apoi se compară al doilea element cu al patrulea, adică 25 cu 300. Acestea sînt în ordine crescătoare și de aceea ele nu se permută. Apoi se compară elementul al treilea cu al cincilea, adică 83 cu 3. Cum  $83 > 3$ , elementele respective se permută și se obține:

4    25    3    300    83    173    95    14    415    256

În acest moment se permută și primul element cu al treilea, deoarece nici acestea nu se află în ordinea cerută. Se obține:

3    25    4    300    83    173    95    14    415    256

La fiecare permutare a două valori, algoritmul compară valoarea mai mică permutată cu cea precedentă (poziția precedentă se stabilește folosind incrementul) și realizează o nouă permutare dacă este nevoie.

Această comparație se continuă, propagindu-se spre începutul șirului, pînă cînd:

a. nu se mai fac permutări;

sau

b. nu mai există elemente în șir (se ajunge la un element care nu mai are precedent în șir).

În continuare se compară elementul al patrulea cu al șaselea, adică 300 cu 173. Acestea se permută:

3    25    4    173    83    300    95    14    415    256

Apoi se compară elementul mai mic permutat (173) cu precedentul lui, conform incrementului, adică cu 25. Cum  $25 < 173$ , aceste elemente nu se permută.

Se continuă parcurgerea șirului, adică se compară elementul al 5-lea cu al 7-lea (83 și 95). Acestea sînt în ordine crescătoare și deci nu se permută. Apoi se continuă cu elementul al 6-lea, care se compară cu al 8-lea: 300 cu 14. Acestea se permută și se obține:

3    25    4    173    83    14    95    300    415    256

Apoi se compară 173 cu 14 și se constată că și aceste elemente trebuie să se permute:

3    25    4    14    83    173    95    300    415    256

Propagarea comparării spre începutul șirului continuă cu încă un pas și anume, se compară 25 cu 14. Cum  $25 > 14$  se permută și aceste elemente obținindu-se:

3    14    4    25    83    173    95    300    415    256

Se continuă cu parcurgerea șirului, comparindu-se elementul al 7-lea cu al 9-lea. Aceste elemente sînt 95 și 415, care sînt în ordine crescătoare, deci ele nu se permută. În sfîrșit, se compară elementul al 8-lea cu al 10-lea și se constată că ele trebuie permutate. Se obține:

3    14    4    25    83    173    95    256    415    300

Compararea elementului al 8-lea cu al 6-lea nu conduce la alte permutări și în felul acesta se încheie cea de a doua parcurgere a șirului. Se înjumătățește incrementul și se obține:

$inc = 1$ .

În acest moment se parcurge șirul comparînd elementele învecinate.

Se compară primul element cu al doilea: 3 cu 14. Ei sînt în ordine crescătoare, deci nu se permută. Se compară 14 cu 4 și cum aceste elemente nu sînt în ordine crescătoare, ele se permută:

3    4    14    25    83    173    95    256    415    300

Apoi se compară elementul al doilea cu precedentul (4 cu 3). Ele sînt în ordinea cerută și nu se mai permută.

O nouă permutare are loc la compararea elementului al 6-lea (173) cu al 7-lea (95). Se obține:

3    4    14    25    83    95    173    256    415    300

Nici această permutare nu se propagă spre începutul șirului deoarece  $83 < 95$ .

Ultima permutare se realizează la compararea ultimelor două elemente: 415 cu 300. După permutarea lor se încheie cea de treia parcurgere a șirului:

3    4    14    25    83    95    173    256    300    415

Se înjumătățește incrementul și se obține  $inc = 0$ . Aceasta corespunde momentului în care șirul este sortat.

Aplicînd metoda bulelor la același exemplu, se constată că sînt necesare mai multe parcurgeri și permutări decît în cazul utilizării metodei Shell.

Eficiența metodei Shell este mult mai mare în comparație cu metoda bulelor în cazul în care se sortează un număr mai mare de date (de ordinul sutelor sau miilor).

O influență asupra eficienței metodei o are și modul în care se alege incrementul. În lucrarea [11] se arată că valorile incrementului la diferite

parcurgeri ale șirului de sortat este bine să fie numere prime între ele.

Pentru alte detalii în legătură cu alegerea incrementului la fiecare pas, vezi [3].

În capitolul de față ne mărginim la alegerea incrementului inițial egal cu  $n/2$ , unde  $n$  este numărul elementelor șirului de sortat. De asemenea după fiecare parcurgere a șirului, acesta se va înjumătăți. Cazul cel mai nefavorabil al acestei metode se obține când  $n$  este o putere a lui doi. Acest caz se poate evita dacă la fiecare parcurgere a șirului incrementul se determină prin relația:

$$\text{increment} = \text{increment}/2 + 1, \text{ dacă } \text{increment} > 2.$$

Metoda de sortare Shell poate fi definită astfel:

1.  $\text{inc} = n/2$ , unde prin  $n$  am notat numărul elementelor care se sortează.
2. Se realizează o parcurgere a șirului de elemente care se sortează.
3.  $\text{inc} = \text{inc}/2$ .
4. Dacă  $\text{inc} > 0$ , atunci se reia de la pasul 2.

Altfel șirul este sortat.

Parcurgerea șirului de elemente implică pașii următori:

1.  $i = \text{inc}$ .
2.  $j = i - \text{inc} + 1$ .
3. Dacă  $j > 0$  și elementele de ordine  $j$  și  $j + \text{inc}$  nu satisfac criteriul de ordonare, atunci elementele respective se permută.

Altfel se continuă cu pasul 6.

4.  $j = j - \text{inc}$ .
5. Se reia de la pasul 3.
6.  $i = i + 1$ .
7. Dacă  $i > n$ , se termină parcurgerea curentă a șirului.

Altfel se reia de la pasul 2.

Pașii 4 și 5 se realizează în cazul în care s-a efectuat permutarea elementelor de ordinul  $j$  și  $j + \text{inc}$ . În acest caz se permută, dacă este necesar, elementele precedente elementului deplasat în poziția  $j$ . Elementele precedente sunt:

$j - \text{inc}, j - 2 * \text{inc}, \dots$



Aceasta se realizează repetind pasul 3 atît timp cît  $j > 0$  și se fac permutări.

Procedind ca în [2], metoda descrisă mai sus se transcrie imediat în limbajul C, ca mai jos.

Presupunem că elementele de sortat se află într-un tablou *tab* de tip *int* și ele sînt în număr de *n*.

```
void shellsort(int tab[],int n)
/* sorteaza in ordine crescatoare elementele lui tab prin metoda Shell */
{
    int inc;
    int i,j,t;

    for(inc = n/2;inc > 0; inc /= 2)

/* se realizeaza o parcurgere a sirului */
        for(i = inc;i < n;i++)

/* - se compara doua elemente aflate la distanta inc unul de altul si daca
    este cazul se permuta;
    - daca s-a realizat o permutare, atunci elementul deplasat in fata
    se compara cu precedentele lui si se permuta daca este cazul. */
            for(j = i-inc;j >= 0 &&
                tab[j] > tab[j+inc];j -= inc){
                t = tab[j];
                tab[j] = tab[j+inc];
                tab[j+inc] = t;
            }
}
```

## 14.2. Sortare rapidă

Această metodă a fost publicată de C.A.R.Hoare în lucrarea [20].

Ideea sortării rapide, ca și sortarea lui Shell, se bazează pe faptul că pentru a atinge o viteză mai bună este de dorit ca să permutăm elementele aflate la o distanță mai mare unul de altul decît 1.

Pentru a ne fixa ideile să presupunem că dorim să sortăm crescător elementele tabloului *tab* de tip *int*, numărul lor fiind *n*.

Alegem, în mod arbitrar, un element al tabloului *tab* și-l atribuim variabilei *x*. Apoi vom parcurge elementele tabloului și vom face permutări de elemente dacă este necesar, așa încît toate elementele mai mici decît *x* să

fie în stînga tabloului (indici mai mici), iar cele mai mari în dreapta lui. În felul acesta, în etapa următoare, se poate considera că avem de sortat două tablouri distincte, unul care conține elemente mai mici decît  $x$  și unul care conține elemente mai mari decît  $x$ . Aceste tablouri se sortează și ele prin același procedeu. Se poate considera că, acest procedeu constă în divizarea unui tablou în două părți, așa încît elementele unuia să fie toate mai mici decît elementele celuilalt. De aceea, această metodă se mai numește și metoda prin *unterschub* de poziți.

Metoda sortării rapide poate fi realizată ca mai jos:

1. Se alege un element oarecare al tabloului, de exemplu elementul aflat la mijlocul tabloului și se atribuie lui  $x$ .
2. Se parcurge tabloul de la stînga la dreapta pînă cînd se găsește un element  $\text{tab}[i] \geq x$ .
3. Se parcurge tabloul de la dreapta la stînga pînă cînd se găsește un element  $\text{tab}[j] \leq x$ .
4. Se permută între ele elementele  $\text{tab}[i]$  cu  $\text{tab}[j]$ , dacă  $i$  nu-l depășește pe  $j$ .
5. Se continuă pașii 2, 3 și 4 de mai sus pînă cînd se ajunge ca  $i > j$

Cînd  $i > j$ , tabloul este divizat în două.

Pașii de mai sus se continuă apoi, asupra fiecăreia dintre cele două părți în care s-a divizat tabloul inițial. În felul acesta se divide fiecare parte în două părți, realizîndu-se o divizare în 4 părți a tabloului inițial și așa mai departe. Divizarea continuă pînă cînd fiecare parte conține un singur element. În acest moment tabloul are elementele sortate în ordine crescătoare.

Evident, se poate defini un algoritm analog pentru a sorta elementele unui tablou în ordine descrescătoare.

Mai jos, exemplificăm metoda sortării rapide folosind tabloul la care s-a aplicat metoda lui Shell:

173   25   4   300   256   83   95   14   415   3

Sortarea se realizează conform pașilor de mai jos:

1.  $\text{inf} = 1$  (indicele primului element).
2.  $\text{sup} = 10$  (indicele ultimului element).
3. Se alege elementul de indice:

$$(\text{inf} + \text{sup})/2$$

deci

$$x = \text{tab}[(1+10)/2] = \text{tab}[5] = 256.$$

4.  $i = \text{inf.}$

5.  $j = \text{sup.}$

6. Se parcurg elementele tabloului de la stnga la dreapta pînă se găsește un element mai mare sau egal cu 256.

Acesta este 300, deci  $i = 4$  și  $\text{tab}[i] = 300$ .

7. Se parcurg elementele tabloului de la dreapta spre stnga pînă cînd se găsește un element mai mic sau egal cu 256.

Acesta este chiar ultimul element.

Deci  $j = 10$  și  $\text{tab}[j] = 3$ .

8. Cum  $i < j$ , se permută  $\text{tab}[i]$  cu  $\text{tab}[j]$ :

173   25   4   3   256   83   95   14   415   300.

9.  $i = i+1$ , deci  $i = 5$ .

10.  $j = j-1$ , deci  $j = 9$ .

Cum  $i < j$ , se reia de la punctul 6:

6. Se obține  $i = 5$ , deoarece  $\text{tab}[5] = 256$  și  $256 \geq 256$  este primul element cu această proprietate.

7. Se obține  $j = 8$ , deoarece  $\text{tab}[8] = 14$  și  $14 < 256$  este primul element întilnit de la dreapta spre stnga care să aibă această proprietate.

8. Cum  $i < j$ , se permută cele două elemente:

173   25   4   3   14   83   95   256   415   300.

9.  $i = i+1$ , deci  $i = 6$ .

10.  $j = j-1$ , deci  $j = 7$ .

Cum  $i < j$  se reia de la punctul 6.

6.  $\text{tab}[8] = 256 \geq 256$  și acesta este primul element cu această proprietate.

$i = 8$ .

7.  $\text{tab}[7] = 95 < 256$ .

$j = 7$ .

8. Cum  $i > j$ , nu se mai fac permutări.

În acest moment s-a terminat procesul de divizare cu  $i = 8$  și  $j = 7$ .

Tabloul s-a divizat în două părți:

- partea întâi:  $\text{tab}[1], \text{tab}[2], \dots, \text{tab}[j] = 7$ ;
- partea a doua:  $\text{tab}[8], \text{tab}[9]$  și  $\text{tab}[10]$ .

În continuare se aplică aceeași procedură la fiecare din aceste două părți pentru prima parte se ia:

$\text{inf} = 1$  și  $\text{sup} = 7$ .

La pasul 3 se obține:

3.  $x = \text{tab}[(1+7)/2] = \text{tab}[4] = 3$ .
4.  $i = 1$ .
5.  $j = 7$ .
6.  $\text{tab}[1]$  este primul element așa încît  $\text{tab}[i] > x$ .  
Deci  $i = 1$ .
7.  $\text{tab}[4]$  este primul element așa încît  $\text{tab}[j] <= x$ .  
Deci  $j = 4$ .
8. Cum  $i < j$ , se permută  $\text{tab}[i]$  cu  $\text{tab}[j]$ :

3    25    4    173    14    83    95.

9.  $i = i + 1$ .  
Deci  $i = 2$ .

10.  $j = j - 1$ .  
Deci  $j = 3$ .

Cum  $i < j$ , procesul se reia de la punctul 6.

6. Deoarece  $\text{tab}[i] = 25 > x = 3$ ,  $i$  nu se modifică.
7.  $\text{tab}[3] = 4$ ,  $\text{tab}[2] = 25$  și  $\text{tab}[1] = 3$  deci  $\text{tab}[1] <= x = 3$  și  $j = 1$ .
8. Cum  $i = 2$  și  $j = 1$ ,  $i > j$  și nu se mai fac permutări.

În acest moment s-a obținut o nouă divizare.

Cele două părți sînt:

- prima parte: 3;

- a doua parte: 25    4    173    14    83    95.

deoarece prima parte s-a redus la un element, se continuă cu partea a doua:

1.  $\text{inf} = 2$ .
2.  $\text{sup} = 7$ .
3.  $x = \text{tab}[(2+7)/2] = \text{tab}[4] = 173$ .
4.  $i = 2$ .
5.  $j = 7$ .
6. Se obține  $i = 4$ , deoarece  $\text{tab}[4] \geq x$  este primul element cu această proprietate.
7.  $j = 7$ , deoarece  $\text{tab}[7] = 95 < x$ .
8.  $i < j$  deci se permută  $\text{tab}[i]$  cu  $\text{tab}[j]$ :

25    4    95    14    83    173.

9.  $i = i + 1$ .

Deci  $i = 5$ .

10.  $j = j - 1$ .

Deci  $j = 6$ .

Cum  $i < j$ , se reia de la pasul 6.

6. Se obține  $i = 7$  deoarece singurul element cu proprietatea  $\text{tab}[i] \geq 173$  este  $\text{tab}[7] = 173$ .
7.  $j$  nu se modifică deoarece  $\text{tab}[6] = 83 < x = 173$ .
8.  $i > j$ , deci elementele respective nu se permută.

S-a obținut o nouă divizare a tabloului:

- prima parte: 25    4    95    14    83;
- partea a doua: 173.

Se continuă cu prima parte:

1.  $\text{inf} = 2$ .
2.  $\text{sup} = 6$ .
3.  $x = \text{tab}[(2+6)/2] = \text{tab}[4] = 95$ .
4.  $i = 2$ .
5.  $j = 6$ .

6.  $i = 4$ , deoarece  $\text{tab}[4] = 95 \geq 95$ .
7.  $j$  rămîne 6, deoarece  $\text{tab}[6] = 83 < 95$ .
8.  $i < j$  și de aceea se permută  $\text{tab}[i]$  cu  $\text{tab}[j]$ :

25    4    83   14    95.

9.  $i = i + 1$ ;  $i = 5$ .
10.  $j = j - 1$ ;  $j = 5$ .

Cum  $i \leq j$  se reia de la punctul 6.

6.  $i = 6$  deoarece  $\text{tab}[6] = 95 \geq 95$ .
7.  $j$  rămîne 5 deoarece  $\text{tab}[5] = 14 < 95$ .
8.  $i > j$ , deci elementele respective nu se permută.

S-a obținut o nouă divizare a tabloului:

- prima parte: 25    4    83   14;
- partea a doua: 95.

Se continuă cu prima parte:

1.  $\text{inf} = 2$ .
2.  $\text{sup} = 5$ .
3.  $x = \text{tab}[(2+5)/2] = \text{tab}[3] = 4$ .
4.  $i = 2$ .
5.  $j = 5$ .
6.  $i$  rămîne pe loc deoarece  $\text{tab}[2] = 25 > x = 4$ .
7.  $j = 3$  deoarece  $\text{tab}[3] = 4 \leq x = 4$ .
8. deoarece  $i < j$ , se face permutare:

4        25   83   14.

9.  $i = i + 1$ , deci  $i = 3$ .
10.  $j = j - 1$ , deci  $j = 2$ .

Deoarece  $i > j$ , s-a obținut o nouă divizare:

- prima parte: 4;
- partea a doua: 25   83    14.

Se continuă cu partea a doua:

1.  $\text{inf} = 3$ .
2.  $\text{sup} = 5$ .
3.  $x = \text{tab}[(3+5)/2] = \text{tab}[4] = 83$ .
4.  $i = 3$ .
5.  $j = 5$ .
6.  $i = 4$  deoarece  $\text{tab}[4] = 83 >= 83$ .  
Este primul element cu această proprietate.
7.  $j$  rămâne pe loc, deoarece  $\text{tab}[5] = 14 < 83$ .
8.  $i < j$ , deci se face permutarea:

25      14    83.

9.  $i = i+1$ , deci  $i = 5$ .
10.  $j = j-1$ , deci  $j = 4$ .

Deoarece  $i > j$ , s-a obținut o nouă divizare:

- prima parte: 25    14;
- partea a doua: 83.

Se continuă cu prima parte:

1.  $\text{inf} = 3$ .
2.  $\text{sup} = 4$ .
3.  $x = \text{tab}[(3+4)/2] = \text{tab}[3] = 25$ .
4.  $i = 3$ .
5.  $j = 4$ .
6.  $i$  rămâne pe loc deoarece  $\text{tab}[3] = 25 >= x = 25$ .
7.  $j$  rămâne pe loc deoarece  $\text{tab}[4] = 14 <= x = 25$ .
8. Cum  $i < j$ , se face permutare:

14      25.

9.  $i = i+1$ , deci  $i = 4$ .
10.  $j = j-1$ , deci  $j = 3$ .

Cum  $i > j$ , s-a obținut o nouă divizare:

- prima parte: 14;

– partea a doua: 25.

În momentul de față tabloul *tab* are elementele ordonate ca mai jos:

3      4    14 25    83    95      173 256    415    300.

El este aproape sortat. Partea netratată este partea a doua de la prima divizare:

256    415    300.

Aplicăm același procedeu:

1.  $\text{inf} = 8$ .
2.  $\text{sup} = 10$ .
3.  $x = \text{tab}[(8+10)/2] = \text{tab}[9] = 415$ .
4.  $i = 8$ .
5.  $j = 10$ .
6.  $i = 9$ , deoarece  $\text{tab}[9] = 415 \geq x = 415$  și acesta este primul element cu această proprietate.
7.  $j$  rămâne pe loc, deoarece  $\text{tab}[10] = 300 < x = 415$ .
8.  $i < j$ , deci se face permutarea:

256    300    415.

9.  $i = i+1, i = 10$ .
10.  $j = j-1, j = 9$ .

Cum  $i > j$ , s-a obținut divizarea:

- prima parte: 256    300;
- partea a doua: 415.

Se continuă cu prima parte:

1.  $\text{inf} = 8$ .
2.  $\text{sup} = 9$ .
3.  $x = \text{tab}[(8+9)/2] = \text{tab}[8] = 256$ .
4.  $i = 8$ .
5.  $j = 9$ .
6.  $i$  rămâne pe loc, deoarece  $\text{tab}[i] = 256 \geq x = 256$ .



7.  $j = 8$ , deoarece  $\text{tab}[j] = 256 \leq 256$  și acesta este primul element cu această proprietate.
8. Cum  $i = j$ , nu se face permutare.
9.  $i = i + 1$ , deci  $i = 9$ .
10.  $j = j - 1$ , deci  $j = 7$ .

Cum  $i > j$ , procesul se încheie, tabloul *tab* fiind sortat în ordine crescătoare:

3      4    14 25    83    95    173 256    300    415.

Din acest exemplu se observă că diviziunea este în mare măsură dependentă de elementul separator ales la fiecare pas. Dacă acesta este chiar elementul minim dintre elementele tabloului supus divizării, atunci prima parte se reduce chiar la acest element. În mod analog, dacă elementul separator este elementul maxim, atunci partea a doua se reduce chiar la acest element.

Acestea sînt cazurile cele mai ineficiente deoarece tabloul se divide în așa fel încît una din părți conține un singur element.

În general, divizările pentru tablouri mari este bine să fie cît mai echilibrate, adică cele două părți să fie de lungimi apropiate. Nu există însă un procedeu simplu care să asigure acest fapt. Alegerea elementului de la mijlocul tabloului este o metodă simplă și destul de practică. Mai jos o să indicăm și alte metode pentru a alege elementul separator.

În continuare, descriem o funcție, în limbajul C, pentru metoda ilustrată mai sus. Numim *quicksort* funcția respectivă. Procedura *quicksort* se descrie simplu dacă funcția respectivă este recursivă. Natura ei recursivă decurge din faptul că ea constă din următoarele:

- se divide tabloul în două părți în așa fel încît elementele unei părți să nu depășească elementele celeilalte părți;
- pentru fiecare parte se aplică aceeași procedură de divizare pînă cînd lungimea părții devine 1.

Tabloul este sortat cînd lungimea fiecărei părți a devenit egală cu 1.

Procedura de divizare constă, așa cum s-a văzut mai sus, din selectarea elementului  $x$  din mijlocul tabloului, iar cele două părți se obțin permutînd elementele mai mari sau egale cu  $x$  din stînga tabloului, cu cele mai mici sau egale cu  $x$  din dreapta tabloului.

Un element  $x[i]$  este în stînga lui  $x[j]$ , dacă  $i < j$ . În mod analog, un element  $x[k]$  este în dreapta lui  $x[j]$ , dacă  $k > j$ .

În anumite cazuri este necesar să se permute chiar și elementul selectat  $x$ , deoarece în caz contrar nu se poate realiza divizarea. Într-adevăr  $x$  nu-și schimbă poziția dacă în stînga lui sînt atîtea elemente mai mari decît el cîte sînt în dreapta lui mai mici decît el. Evident, această condiție nu este îndeplinită frecvent și de aici rezultă necesitatea permutării lui.

Funcția *quicksort* are parametri:

- |            |   |
|------------|---|
| <i>tab</i> | - Tabloul care se sortează.                   |
| <i>inf</i> | - Indicele inferior al părții care se divide. |
| <i>sup</i> | - Indicele superior al părții care se divide. |

Dacă *tab* are  $n$  elemente, atunci funcția *quicksort* se apelează astfel:

```
quicksort(tab,0,n-1);
```

Definim mai jos funcția *quicksort*.

```
void quicksort(int tab[],int inf,int sup)
/*- sorteaza in ordine crescatoare tabloul tab, folosind metoda sortarii
   rapide;
   - elementul separator este elementul din mijlocul tabloului. */
{
    int i,j,x,t;

    i = inf;
    j = sup;

    /* se alege elementul separator */
    x = tab[(i+j)/2];

    /* se face divizarea */
    do{
        /* avans peste elementele din stînga lui x si mai mici decît el */
        while(i < sup && tab[i] < x)
            i++;

        /* avans peste elementele din dreapta lui x mai mari decît el */
        while(j > inf && tab[j] > x)
            j--;

        if(i <= j){
            if(i < j){ /* daca i < j se permuta elementele */
                t = tab[i];
                tab[i] = tab[j];
                tab[j] = t;
            }
        }
    }
```

```
/* se trece la elementele urmatoare */
```

```
    i++;
```

```
    j--;
```

```
}
```

```
/* daca i > j s-a realizat divizarea:
```

```
    - prima parte:
```

```
        inf - limita inferioara;
```

```
        j - limita superioara;
```

```
    - partea a doua:
```

```
        i - limita inferioara;
```

```
        sup - limita superioara;
```

```
    daca i <= j, procesul de divizare continua. */
```

```
while(i <= j);
```

```
/* daca inf < j, se trece la divizarea primei parti deoarece aceasta contine  
mai mult decit un element */
```

```
if (inf < j)
```

```
    quicksort(tab, inf, j);
```

```
/*daca i < sup se trece la divizarea partii a doua deoarece contine mai  
mult decit un element */
```

```
if (i < sup)
```

```
    quicksort(tab, i, sup);
```

```
}
```

Revenim la problema determinării elementului separator. Pentru detalii, cititorul poate consulta lucrările [3] și [11].

O metodă relativ simplă și care dă adesea rezultate bune constă în alegerea *medianei* următoarelor elemente:

- primul element al tabloului;
- elementul aflat la mijlocul tabloului;
- ultimul element al tabloului.

Amintim că dacă  $a, b, c$  sînt trei numere și între ele există relația

$$a < b < c$$

atunci *mediana* lor este  $b$ .

În felul acesta se elimină situațiile în care elementul separator este elementul minim sau maxim al tabloului.

Propunem cititorului să modifice funcția *quicksort* în așa fel încît

elementul separator să fie ales ca mediană a celor trei elemente amintite mai sus.

O altă metodă, descrisă amănunțit în [11] constă în alegerea ca element separator a *primului element al tabloului* și deplasarea acestuia pe locul lui final în procesul divizării tabloului.

În stînga lui se deplasează elementele mai mici decît el, iar în dreapta cele mai mari decît el. În felul acesta tabloul este divizat în două părți, iar elementul separator, fiind pe locul lui final, nu aparține nici uneia din cele două părți pe care le separă.

Mai jos definim funcția *quicksort* care utilizează această metodă la alegerea elementului separator.

În principiu, această metodă constă în următoarele:

Notăm cu *inf* și *sup* ( $inf < sup$ ) indicii care definesc limita inferioară, respectiv superioară a părții din tabloul *tab* care urmează a fi divizată.

1.  $x = tab[inf]$ .
2.  $i = inf$ .
3.  $j = sup$ .
4. Dacă  $i < j$  se trece la pasul 5.  
Altfel la pasul 9.
5. Se determină un element  $tab[i]$ , incrementînd pe  $i$  așa încît  $tab[i] > x$  și  $i \leq sup$ .
6. Se determină un element  $tab[j]$ , decrementînd pe  $j$ , așa încît  $tab[j] \leq x$ .
7. Dacă  $i < j$ , atunci se permută  $tab[i]$  cu  $tab[j]$ .  
Altfel se trece la pasul 8.
8. Se trece la pasul 4.
9. Se deplasează  $tab[inf]$  pe locul lui final permutîndu-l cu  $tab[j]$ .

În acest moment tabloul este divizat astfel:

– prima parte:

$inf$  - limita inferioară;

$j-1$  - limita superioară.

–  $tab[j]$ :

    are poziția sa finală și nu se va mai permuta.

– partea a doua:

$j+1$  - limita inferioară;

sup - limita superioară.

Procesul se reia pentru părțile care nu s-au redus încă la un singur element (limita inferioară a părții respective este mai mică decât cea superioară).

Exemplificăm metoda pe același exemplu:

173    25    4    300    256    83    95    14    415    3

inf = 1 și sup = 10

1.  $x = \text{tab}[1] = 173$ .
2.  $i = \text{inf} = 1$ .
3.  $j = \text{sup} = 10$ .
4.  $i < j$ , se trece la pasul 5.
5.  $\text{tab}[i] > 173$  pentru  $i = 4$ :  $\text{tab}[4] = 300$ .
6.  $\text{tab}[j] < 173$  pentru  $j = 10$ :  $\text{tab}[10] = 3$ .
7. Cum  $i = 4 < j = 10$ , se permută cele două elemente:

173    25    4    3    256    83    95    14    415    300.

8. Se reia de la pasul 4.
4.  $i < j$ , se trece la pasul 5.
5.  $\text{tab}[5] = 256 > 173$ , deci  $i = 5$ .
6.  $\text{tab}[8] = 14 < 173$ , deci  $j = 8$ .
7.  $i < j$ , deci se permută cele 2 elemente:

173    25    4    3    14    83    95    256    415    300.

8. Se reia de la pasul 4.
4.  $i < j$ , se trece la pasul 5.
5.  $\text{tab}[8] > 173$ , deci  $i = 8$ .
6.  $\text{tab}[7] < 173$ , deci  $j = 7$ .
7. Cum  $i > j$ , nu se face permutare.
8. Se reia de la pasul 4.

4. Cum  $i > j$ , se trece la pasul 9.
9. Se permută  $\text{tab}[\text{inf}] = \text{tab}[1] = 173$ , cu  $\text{tab}[j] = \text{tab}[7] = 95$ :  
     95     25   4   3     14     83     173   256   415   300.

Tabloul se descompune în:

- prima parte: A   95   25   4   3     14           83;
- 173 ocupă locul lui final;
- partea a doua: B   256   415   300.

Aplicăm același algoritm la partea A:

$\text{inf} = 1, \text{sup} = 6$

1.  $x = \text{tab}[\text{inf}] = 95$ .
2.  $i = 1$ .
3.  $j = 6$ .
4.  $i < j$ , se trece la pasul 5.
5. Deoarece  $\text{tab}[i] = 95$  este elementul maxim,  $i$  crește până la valoarea maximă, adică  $i$  devine egal cu 6.
6.  $\text{tab}[j] = 83 < x = 95$ , înseamnă că  $j$  rămâne nemodificat.
7.  $i = j = 6$ , deci nu se face permutare.
8. Se reia de la pasul 4.
4.  $i = j$ , se trece la pasul 9.
9. Se permută  $\text{tab}[\text{inf}] = \text{tab}[1] = 95$  cu  $\text{tab}[j] = 83$ :

    83     25   4   3     14     95.

Tabloul se descompune în:

- prima parte: AA   83   25   4   3     14;
- 95 ocupă locul lui final;
- partea a doua: vidă.

Aplicăm același algoritm la partea AA:

$\text{inf} = 1, \text{sup} = 5$

1.  $x = \text{tab}[\text{inf}] = 83$ .
2.  $i = 1$ .

3.  $j = 5$ .
4.  $i < j$ , se trece la pasul 5.
5. Deoarece  $\text{tab}[i] = 83$  este elementul maxim,  $i$  crește până la valoarea maximă, adică  $i$  devine egal cu 5.
6.  $\text{tab}[j] = 14 < x = 83$ , înseamnă că  $j$  rămâne nemodificat.
7.  $i = j = 5$ , deci nu se face permutare.
8. Se reia de la pasul 4.
4.  $i = j$ , se trece la pasul 9.
9. Se permută  $\text{tab}[\text{inf}] = \text{tab}[1] = 83$  cu  $\text{tab}[j] = 14$ :  
 $14 \quad 25 \quad 4 \quad 3 \quad 83$ .

Tabloul se descompune în:

- prima parte: AAA  $14 \quad 25 \quad 4 \quad 3$ ;
- 83 ocupă locul lui final;
- partea a doua: vidă.

Aplicăm același algoritm la partea AAA:

$\text{inf} = 1, \text{sup} = 4$

1.  $x = \text{tab}[\text{inf}] = \text{tab}[1] = 14$ .
2.  $i = 1$ .
3.  $j = 4$ .
4.  $i < j$ , se trece la pasul 5.
5.  $i$  crește la valoarea 2, deoarece  $\text{tab}[2] = 25 > x$ .
6.  $j$  nu se modifică deoarece  $\text{tab}[j] = \text{tab}[4] = 3 < x$ .
7.  $i < j$ , se permută  $\text{tab}[2]$  cu  $\text{tab}[4]$ :  
 $14 \quad 3 \quad 4 \quad 25$ .
8. Se reia de la pasul 4.
4.  $i < j$ , se trece la pasul 5.
5.  $i$  crește până la 4, deoarece  $\text{tab}[4] = 25 > x$ .
6.  $j$  devine egal cu 3, deoarece  $\text{tab}[3] = 4 < x$ .
7.  $i > j$ , deci nu se face permutare.

8. Se trece la pasul 4.
4.  $i > j$ , deci se trece la pasul 9.
9. Se permută  $\text{tab}[\text{inf}] = \text{tab}[1] = 14$  cu  $\text{tab}[j] = \text{tab}[3] = 4$ :  
     4      3    14 25.

Tabloul se descompune în:

- prima parte: AAAA    4    3;
- 14 ocupă locul lui final;
- partea a doua: 25.

Aplicăm același algoritm la partea AAAA:

$\text{inf} = 1, \text{sup} = 2$

1.  $x = \text{tab}[\text{inf}] = 4$ .
2.  $i = 1$ .
3.  $j = 2$ .
4.  $i < j$ , se trece la pasul 5.
5. Deoarece  $\text{tab}[i] = 4$  este elementul maxim,  $i$  crește pînă la valoarea maximă, adică  $i$  devine egal cu 2.
6.  $\text{tab}[j] = \text{tab}[2] = 3 < x = 4$ .  
     Înseamnă că  $j$  rămîne nemodificat.
7.  $i = j = 2$ , deci nu se face permutare.
8. Se reia de la pasul 4.
4.  $i = j$ , se trece la pasul 9.
9. Se permută  $\text{tab}[\text{inf}]$  cu  $\text{tab}[j]$ :  
     3      4.

Tabloul se descompune în:

- prima parte: 3;
- 4 ocupă locul lui final;
- partea a doua: vidă.

În acest moment elementele tabloului sînt ordonate astfel:

3      4    14 25    83    95    173 256    415    300.



Ultimele 3 elemente formează partea B de la prima diviziune a aceleiași metode:

$\text{inf} = 8, \text{sup} = 10$

1.  $x = \text{tab}[\text{inf}] = \text{tab}[8] = 256$ .
2.  $i = 8$ .
3.  $j = 10$ .
4.  $i < j$ , se trece la pasul 5.
5.  $i$  devine egal cu 9 deoarece  $\text{tab}[9] = 415 > x$ .
6.  $j$  devine egal cu 8 deoarece  $\text{tab}[10] > x$  și  $\text{tab}[9] > x$ ,  $\text{tab}[8]$  fiind primul care nu are această proprietate.
7.  $i > j$ , deci nu se face permutare.
8. Se reia de la pasul 4.
4.  $i > j$ , se trece la pasul 9.
9. Cum  $\text{inf} = j$ , nu se face permutare.

Tabloul se descompune în:

- prima parte: vidă;
- 256 ocupă locul lui final;
- partea a doua: BB      415   300;

Aplicăm același algoritm la partea BB și în final se obține:

3      4    14 25    83    95    173 256    300    415.

Mai jos, transcriem acest procedeu în limbajul C.

```
void quicksortinf(int tab[],int inf,int sup)
/* - sorteaza in ordine crescatoare tabloul tab folosind metoda sortarii
   rapide;
   - elementul separator este tab[inf] care se deplasează pe locul lui
   final. */
{
    int i,j,t,x;

    x = tab[inf];
    i = inf;
    j = sup;

    /* divizarea tabloului */
```

```

while(i < j){
/* - se determina cel mai mic indice i astfel ca tab[i] > x și i <= sup;
- se avanseaza peste elementele mai mici sau egale cu x;
- i = sup daca nu exista un astfel de element. */
    while(tab[i] <= x && i < sup)
        i++;

/* - se determina cel mai mare indice j astfel ca tab[j] <= x;
- se avanseaza peste elementele mai mari decit x. */
    while(tab[j] > x)
        j--;

/* daca i < j, se permuta tab[i] cu tab[j] */
    if(i < j){ /* se face permutare */
        t = tab[i];
        tab[i] = tab[j];
        tab[j] = t;
    }
}

/* - s-a terminat divizarea tabloului;
- tab[inf] se deplaseaza pe locul lui final care este j;
- totodata tab[j] poate fi deplasat in pozitia inf;
- deci se permuta tab[inf] cu tab[j];
- permutarea nu are loc daca j = inf. */
if(j > inf){
    tab[inf] = tab[j];
    tab[j] = x;
}

/* s-a realizat divizarea:
- prima parte:
    tab[inf], tab[inf+1], ..., tab[j-1];
- tab[j] ocupa pozitia lui finala;
- partea doua:
    tab[j+1], tab[j+2], ..., tab[sup]. */
if(inf < j-1)

/* prima parte are cel putin 2 elemente */
quicksortinf(tab, inf, j-1);

if(j+1 < sup)

```

```
/* partea a doua are cel putin 2 elemente */
    quicksortinf(tab, j+1, sup);
}
```

Menționăm că în lucrarea [11], această metodă este transcrisă în limbajul C într-o funcție nerecursivă.

### Exerciții:

14.1 Să se scrie un program care realizează următoarele:

- citește o succesiune de numere separate prin caractere albe și precedate de numărul lor;
- sortează în ordine crescătoare numerele citite și apoi le afișează; sortarea se realizează folosind următoarele metode:
  - metoda bulelor;
  - sortare Shell;
  - sortare rapidă cu alegerea elementului din mijloc ca element separator;
  - sortare rapidă cu alegerea primului element ca element separator.
- afișează numărul permutărilor efectuate în cadrul fiecărei metode de sortare utilizate.

### PROGRAMUL BXIV1

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "BVIII2.CPP" /* pcit_int */
#include "BVIII3.CPP" /* pcit_int_lim */
#include "BVIII8.CPP" /* pndcit */
#include "BVIII9.CPP" /* pdvcit */

#define MAX 100

void dperm(double *p1, double *p2)
/* permuta valorile de tip double spre care pointeaza p1 si p2 */
{
    double temp;
```

```

temp = *p1;
*p1 = *p2;
*p2 = temp;
}

void dcoPIaza(double tabl[],double tab2[],int n)
/* copiaza primele n elemente din tabloul tab2 in tabloul tab1 */
{
    int i;

    for(i=0;i < n;i++)
        tabl[i]=tab2[i];
}

void afistab(double tab[],int n)
/* - afiseaza primele n elemente din tabloul tab;
   - pe o linie se afiseaza 5 elemente. */
{
    char text[] = "Actionati o tasta pentru a\
                  continua\n";
    int i,j;
    double d;

    printf("\n");
    j=1;
    for(i = 0;i < n;i++){
        d = tab[i];
        printf("%f ",d);
        if(i%5==4){
            printf("\n");
            if(j++ %22==0){
                printf(text);
                getch();
            }
        }
    }
    printf(text);
    getch();
}

void afisantet(char *p) /* afiseaza un antet */
{
    printf("\n\t\t\t%s\n",p);
}

```

```

printf("Actionati o tasta pentru a continua\n");
getch();
}

void afisnrperm(char *p, long n)
/* afiseaza textul spre care pointeaza p si valoarea lui n */
{
    printf("\n Metoda %s\t numar permutari=%ld\n",
           p, n);
}

long bubblesort(double tab[], int n)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind
   metoda bulelor;
   - returneaza numarul permutarilor efectuate. */
{
    int ind, i;
    long s;

    ind = 1;
    s = 0;
    while(ind){
        ind = 0;
        for(i = 0; i < n-1; i++){
            if(tab[i] > tab[i+1]){
                dperm(&tab[i], &tab[i+1]);
                s++; /* numara permutari */
                ind = 1;
            } /* sfirsit if */
        } /* sfirsit while */
        return s;
    } /* sfirsit bubblesort */

long shellsort(double tab[], int n)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind
   metoda lui Shell;
   - returneaza numarul permutarilor efectuate. */
{
    int inc;
    int i, j;
    long s;

    s = 0;

```

```

for(inc = n/2; inc > 0; inc /= 2)
    for(i = inc; i < n; i++)
        for(j = i-inc; j >= 0 &&
            tab[j] > tab[j+inc]; j -= inc){
            dperm(&tab[j], &tab[j+inc]);
            s++;
        }
    return s;
} /* sfirsit shellsort */

void quicksort(double tab[], int inf, int sup)
/*- sorteaza elementele tabloului tab in ordine crescatoare folosind
    metoda sortarii rapide;
- elementul separator este elementul din mijlocul tabloului;
- numarul permutarilor realizate se atribuie variabilei globale s. */
{
    extern long s;
    int i, j;
    double x;

    i = inf;
    j = sup;
    x = tab[(i+j)/2];
    do{
        while(i < sup && tab[i] < x)
            i++;
        while(j > inf && tab[j] > x)
            j--;
        if(i <= j){
            if(i < j){
                dperm(&tab[i], &tab[j]);
                s++;
            }
            i++;
            j--;
        }
    }while(i <= j);
    if(inf < j)
        quicksort(tab, inf, j);
    if(i < sup)
        quicksort(tab, i, sup);
} /* sfirsit quicksort */

```

```

void quicksortinf(double tab[],int inf,int sup)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind
   metoda sortarii rapide;
   - elementul separator este primul element al tabloului si acesta se
   deplaseaza pe locul lui final;
   - numarul permutarilor realizate se atribuie variabilei globale s. */
{
    extern long s;
    int i,j;
    double x;

    x = tab[inf];
    i = inf;
    j = sup;
    while(i < j){
        while(tab[i] <= x && i < sup)
            i++;
        while(tab[j] > x)
            j--;
        if(i < j){
            dperm(&tab[i],&tab[j]);
            s++;
        }
    }
    if(j > inf){
        tab[inf] = tab[j];
        tab[j] = x;
        s++;
    }
    if(inf < j-1)
        quicksortinf(tab,inf,j-1);
    if(j+1 < sup)
        quicksortinf(tab,j+1,sup);
} /* sfirsit quicksortinf */

long s;

main()
/* - citeste o succesiune de numere separate prin caractere albe si
   precedate de numarul lor;
   - sorteaza in ordine crescatoare numerele citite si apoi le afiseaza;
   - sortarea se face folosind urmatoarele metode:
       metoda bulelor;

```

```

        sortare shell;
        sortare rapida cu alegerea elementului din mijloc ca separator
        sortare rapida cu alegerea primului element ca element separator
- afiseaza numarul permutarilor efectuate in cadrul fiecarei metode
  utilizate. */
{
    double tini[MAX],tsort[MAX];
    int n;
    long a,b,c,d;

/* citeste numerele de sortat */
    if((n = pdvcit(MAX,tini)) < 2){
        printf("nu s-au tastat nici 2 numere\n");
        exit(1);
    }

/* sortare prin metoda bulelor */
    dcopiaza(tsort,tini,n); /* tsort = tini */
    a = bubblesort(tsort,n);
    afisantet("metoda bulelor");
    afistab(tsort,n);

/* sortare prin metoda Shell */
    dcopiaza(tsort,tini,n);
    b = shellsort(tsort,n);
    afisantet("metoda Shell");
    afistab(tsort,n);

/* sortare rapida - elementul separator este elementul din mijlocul
   tabloului */
    dcopiaza(tsort,tini,n);
    s = 0;
    quicksort(tsort,0,n-1);
    c = s;
    afisantet("sortare rapida: elementul din mijloc");
    afistab(tsort,n);

/* sortare rapida - elementul separator este primul element */
    dcopiaza(tsort,tini,n);
    s = 0;
    quicksortinf(tsort,0,n-1);
    d = s;
    afisantet("sortare rapida: primul element");

```



```
afistab(tsort,n);

/* se afiseaza numarul permutarilor */
afisnrperm("Bulelor",a);
afisnrperm("Shell",b);
afisnrperm("Sortare rapida: mijloc",c);
afisnrperm("Sortare rapida: inferior",d);
}
```

## 15. DIN NOU DESPRE PREPROCESARE ÎN C

În capitolul 1 s-a arătat că preprocesarea este o fază care precede compilarea propriu-zisă. Preprocesorul limbajului C este relativ simplu. El, în principiu, execută *substituții* de texte. Prin intermediul lui se pot realiza:

- includeri de fișiere sursă;
- definiții și apeluri de macrouri;
- compilare condiționată.

Preprocesorul recunoaște construcțiile care încep prin caracterul *diez* (#) Acestea se mai numesc și *directive*.

Așa de exemplu, includerile de fișiere se realizează cu ajutorul construcției *#include*. Această construcție a fost descrisă în capitolul 1.

În capitolul de față vom reveni asupra construcției *#define*.

În general, construcția *#define* se folosește la definirea de macrouri.

Apoi, ne vom ocupa de compilarea condiționată.

### 15.1. Definiții și apeluri de macrouri

Definiția constantelor simbolice este un caz particular de definiție de macro.

Amintim că o constantă simbolică se definește printr-o construcție de forma:

***#define nume succesiune de caractere***

Efectul acestei construcții constă în substituirea în textul programului a numelui aflat după *#define* cu *succesiune de caractere* din construcția *#define* respectivă.

Substituția se realizează pentru orice apariție a lui *nume* în textul sursă care urmează după definirea lui prin construcția *#define*, exceptând cazurile în care *nume* apare într-un șir de caractere sau într-un comentariu.

Efectul construcției *#define* se anulează la întâlnirea construcției:

***#undef nume***

După o astfel de construcție, *nume* poate fi redefinit printr-o altă

construcție *#define*.

La scrierea construcției *#define*, *nume* este precedat și urmat de cel puțin un spațiu. Succesiunea de caractere care se substituie numelui începe cu primul caracter care nu este alb. Ea se poate continua pe rîndul următor terminînd-o prin caracterul backslash (\).

Un *macro* este o facilitate generală de prelucrare a textelor prin substituție. Un *macro*, ca și o funcție, are o definiție și unul sau mai multe apeluri. Definiția *macro*ului trebuie să preceadă orice apel al său. Ea definește textul care urmează a se substitui la fiecare apel al *macro*ului. Acest text poate fi variabil, variînd de la apel la apel. De aceea, un astfel de text, de obicei, va conține parametri. Aceștia se concretizează la fiecare apel. Parametri utilizați în definiția *macro*ului se numesc *formali*. Valorile lor dintr-un apel al *macro*ului sînt parametri *efecivi* sau *concreți*.

Forma generală a unei definiții de *macro* este următoarea:

*#define* *nume*(*p1,p2,...,pn*) *text*

unde:

- |                     |   |
|---------------------|---|
| <i>nume</i>         | - Este numele <i>macro</i> ului.  |
| <i>p1,p2,...,pn</i> | - Sînt parametri <i>formali</i> ai <i>macro</i> ului.   |
| <i>text</i>         | - Este textul de substituție.   |
|                     | - El conține parametri <i>formali</i> <i>p1,p2,...,pn</i> .                                       |
|                     | - Dacă este nevoie, el se poate continua pe rîndul următor folosind caracterul <i>backslash</i> . |

Fiecare *p1,p2,...,pn* este un *nume*.

Dacă *macro*ul nu are parametri, atunci este absentă toată construcția (*p1,p2,...,pn*) și evident, textul de substituție devine constant (nu mai conține parametri). În acest caz definiția de *macro* devine o definiție de constantă simbolică.

În scrierea unei definiții de *macro* cu parametri trebuie ca între numele *macro*ului și paranteza deschisă să *nu* existe nici un caracter alb. În caz contrar, *macro*ul se consideră fără parametri și paranteza închisă, ce conține lista parametrilor, se consideră ca făcînd parte din textul de substituție.

Apelul unui *macro* constă din numele lui urmat de lista valorilor parametrilor inclusă între paranteze rotunde.

Parametri de la apel (efecivi) se separă prin virgulă dacă sînt mai mulți. Ei se substituie parametrilor *formali* în textul de substituție al *macro*ului apelat și apoi textul rezultat se substituie apelului.

Correspondența dintre parametri formali și valorile lor de la apel se realizează prin poziție.

Substituirea apelului unui macro prin textul de substituție după ce, în prealabil, eventualii parametri formali au fost înlocuiți cu valorile lor din apel, se numește *expandare*.

Expandarea poate fi anihilată cu ajutorul construcției *#undef*:

**#undef nume**

unde:

*nume* - Este numele macroului.

O definiție de macro este valabilă din punctul în care este scrisă și până la sfârșitul textului sursă în care apare sau până la întâlnirea construcției *#undef* care o anulează.

Apelul unui macro se expandează dacă acesta se află în textul sursă în zona în care este valabilă definiția lui. Evident, un apel de macro nu se expandează dacă se află într-un comentariu sau într-un șir de caractere.

### Exemple:

1. Macroul de mai jos definește maximumul dintre valorile a două expresii. În acest scop se folosește o expresie condițională.

Macroul se definește în felul următor:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Dăm mai jos exemple de apeluri ale acestui macro:

a.

```
int i,j,k;
```

```
...
```

```
k = MAX(i+j,i-j);
```

Apelul MAX(i+j,i-j) se expandează la preprocesare, iar la compilare se întâlnește instrucțiunea de atribuire de mai jos:

```
k = ((i+j) > (i-j) ? (i+j) : (i-j));
```

b.

```
float a,b,c;
```

```
...
```

```
c = MAX(a,b);
```

Rezultatul expandării este:

`c = ((a) > (b) ? (a) : (b));`

2. Mai jos, se definește un macro pentru calculul valorii absolute:

`#define ABS(x) ((x) < 0 ? -(x) : (x))`

Exemple de apeluri și expandări ale acestui macro:

- a.

`int i,j,k;`

`...`

`k = ABS(i-j);`

Se expandează astfel:

`k = ((i-j) < 0 ? -(i-j) : (i-j));`

- b.

`double a,b,c,eps;`

`...`

`c = ABS(a-b) < eps;`

Se expandează astfel:

`c = ((a-b) < 0 ? -(a-b) : (a-b)) < eps;`

3. Macrourele de mai jos se utilizează la transformarea literelor mici în litere mari și invers:

`#define UPPER(c) ((c)-'a'+'A')`

`#define LOWER(c) ((c)-'A'+'a')`

Exemple de apel:

`putchar(UPPER(c));` scrie o literă mare dacă valoarea lui `c` este codul ASCII al unei litere mici;

`putchar(LOWER(c));` scrie o literă mică dacă valoarea lui `c` este codul ASCII al unei litere mari.

4. Macroul de mai jos permite permutarea a două valori de tip `int`:

```
#define IPERM(X,Y) {\n    int t;\n    t = X;\n    X = Y;\n    Y = t;\n}
```

Apelul:

**IPERM (a,b)**

se expandează astfel:

```
{  
    int t;  
  
    t = a;  
    a = b;  
    b = t;  
}
```

Putem realiza un macro mai general de permutare care să fie valabil pentru date numerice de orice tip predefinit:

```
#define PERM(TIP,X,Y) {\nTIP t;\nt = X;\nX = Y;\nY = t;\n}
```

Apelul

**PERM(int,a,b)**

se expandează la fel ca și **IPERM**.

Pentru a permuta date flotante vom folosi apelul:

**PERM(float,a,b)**

Se observă o analogie între funcții și macroui. Ambele au o definiție și unul sau mai multe apeluri. De asemenea, atât în definiția de funcție cât și în cea de macro se pot utiliza parametri formali. În ambele cazuri, valorile parametrilor formali se definesc la apeluri. Există însă o diferență esențială între cele două moduri de definire a valorilor parametrilor.

În cazul macrouilor, valoarea parametrului formal este succesiunea de caractere corespunzătoare lui din apel. Această succesiune de caractere se substituie în locul parametrului formal peste tot unde acesta apare în textul de substituție al macroului. De exemplu, la apelul **PERM(int,a,b)** de mai sus, parametrul formal **TIP** se substituie prin succesiunea de caractere *int* în textul de substituție și în felul acesta se obține declarația:

```
int t;
```

În cazul funcțiilor, parametrilor formali li se atribuie valorile parametrilor de la apel care sînt date de tipuri predefinite sau adrese de zone de memorie.

O altă diferență esențială constă în aceea că, în timp ce apelul unei funcții înseamnă un salt la o zonă de memorie în care se păstrează instrucțiunile în format executabil corespunzător funcției respective, în cazul macrouilor apelul unui macro se înlocuiește chiar cu instrucțiunile rezultate în urma compilării, expandării macroului respectiv.

De exemplu, fie funcția:

```
int abs(int x) /* returneaza valoarea absoluta a lui x */
{
    return x < 0 ? -x : x;
}
```

și apelul ei:

```
int i,j;
...
j = abs(i);
...
```

La apelul funcției se realizează un salt la zona în care se află rezultatul compilării funcției *abs* definită mai sus.

După executarea instrucțiunilor corespunzătoare funcției *abs*, se revine în punctul de apel și se atribuie lui *j* valoarea returnată de funcție (valoarea absolută a lui *i*).

Înlocuind definiția funcției *abs* prin definiția de macro de mai jos:

```
#define abs(x) ((x) < 0 ? -(x) : (x))
```

același apel se expandează astfel:

```
j = ((i) < 0 ? -(i) : (i));
```

În acest caz, apelul macroului se înlocuiește prin instrucțiunile care calculează valoarea absolută a lui *i*.

Cu alte cuvinte, în cazul macrouilor instrucțiunile definite de ele se generează și apoi se compilează direct în locurile apelurilor, adică în poziția în care este nevoie de ele. În cazul funcțiilor, instrucțiunile definite de corpul unei funcții se păstrează într-o zonă de memorie a cărei poziție nu are nimic comun cu apelurile funcției. La fiecare apel al funcției se realizează un salt la zona respectivă, iar după terminarea execuției funcției se revine la punctul de după apel.

Având în vedere cele de mai sus, se pune problema existenței avantajelor la utilizarea macrouilor în locul funcțiilor.

Utilizarea macrouilor este avantajoasă pentru procese de calcul foarte

simple, cum sînt exemplele de mai sus: maximul dintre două valori numerice, valoarea absolută, permutarea valorilor a două variabile etc.

Într-adevăr, prin utilizarea macrouilor, în locul funcțiilor, se elimină apelurile funcțiilor. Amintim că la apelul unei funcții se face nu numai salt la zona de memorie corespunzătoare funcției apelate, ci se alocă pe stivă parametri formali și variabilele automate ale funcției respective. De asemenea, la revenirea din funcție se face curățirea stivei. Toate aceste activități pot fi mai complexe decît însuși procesul de calcul realizat prin funcție. De aceea, în acest caz, se preferă utilizarea macrouilor în locul funcțiilor.

Prin utilizarea macrouilor nu se mai face apel și nici alocări de parametri pe stivă și respectiv dezalocarea lor. Apelul macroului pur și simplu se înlocuiește prin instrucțiunile care realizează procesul de calcul exprimat prin expandarea macroului. În acest caz se obișnuiește să se spună că macroul definește o funcție generată *în linie* (*in-line*).

Pentru procese de calcul mai complexe nu se justifică utilizarea macrouilor, deoarece substituirea fiecărui apel prin textul de substituție al macroului apelat, va conduce la un consum mare de memorie.

În limbajul C++ se pot defini funcții *in-line*. Acestea, ca și macrouile, se generează pe locul apelurilor lor. De aceea, o funcție *in-line* trebuie să corespundă unui calcul simplu care se poate realiza cu un număr mic de instrucțiuni (2-3 instrucțiuni). În caz contrar, se poate ajunge la un consum mare de memorie.

Diferența dintre funcțiile *in-line* și macroui constă în faptul că, la apelul funcțiilor *in-line* se fac controale asupra tipului parametrilor utilizați la apel, în timp ce în cazul macrouilor astfel de controale nu mai au loc.

## 15.2. Compilare condiționată

Compilarea condiționată permite să se aleagă, dintr-un text general, părțile care să se compileze împreună. Acest lucru este util pentru programe care au părți comune sau care depind de calculator sau de versiunea compilatorului. În felul acesta, în funcție de anumiți parametri, se pot alege părțile din textul sursă care urmează a fi compilate împreună.

Compilarea condiționată se realizează folosind construcțiile (directivele) preprocesorului:

**#if, #ifdef și #ifndef.**

Construcția **#if** are două forme:



```
#if expr
    text
#endif
```

și

```
#if expr
    text1
#else
    text2
#endif
```

unde:

- expr* - Este o expresie constantă (valoarea ei poate fi evaluată de preprocesor la întâlnirea ei).
- text*, *text1* și *text2* - Sint texte sursă care sint supuse compilării în funcție de valoarea expresiei *expr*.

În primul format, dacă *expr* are valoarea adevărat (este diferită de zero), atunci *text* se supune preprocesării și în consecință și compilării, apoi se continuă cu textul sursă aflat după *#endif*. În caz contrar (*expr* are valoarea zero), se continuă cu textul aflat după *#endif*.

În formatul al doilea, dacă *expr* are valoarea adevărat, atunci se supune preprocesării *text1* și deci și compilării apoi se continuă cu textul care urmează după *#endif*. În caz contrar (*expr* are valoarea zero) se supune preprocesării *text2* și deci și compilării, apoi se continuă cu textul aflat după *#endif*.

Directivele *#ifdef* și *#ifndef* se utilizează în mod analog.

În cazul directivei *#ifdef* se utilizează următoarele formate:

```
#ifdef nume
    text
#endif
```

și

```
#ifdef nume
    text1
#else
    text2
#endif
```

În primul format, *text* se supune preprocesării și deci și compilării, dacă *nume* este definit în momentul întâlnirii directivei *#ifdef* de către preprocesor.

În formatul al doilea *text1* se preprocesează și apoi compilează numai

dacă *nume* este definit în momentul întâlnirii directivei *#ifdef* de către preprocesor. În caz contrar (*nume* nu este în prealabil definit) se preprocesează și apoi compilează *text2*.

Directiva *#ifndef* are o acțiune opusă directivei *#ifdef*. Deci, *text* din formatul:

```
#ifndef nume  
    text  
#endif
```

se preprocesează și apoi compilează dacă *nume* nu este definit în momentul întâlnirii directivei *#ifndef* de către preprocesor.

În mod analog, formatul:

```
#ifndef nume  
    text1  
#else  
    text2  
#endif
```

realizează preprocesarea și apoi compilarea lui *text1* dacă *nume* nu este definit în momentul întâlnirii directivei *#ifndef* de către preprocesor. În caz contrar (*nume* este în prealabil definit) se preprocesează și apoi compilează *text2*.

### Exemplu:

Un program se compune din două fișiere *fis1.cpp* și *fis2.cpp*. Ambele fișiere conțin apeluri la funcții care au prototipurile în fișierul *stdio.h*.

Întrucât cele două fișiere au fost editate de programatori diferiți, ele au fost la început compilate separat pentru a se elimina erorile sintactice. De aceea, fiecare fișier conține la începutul lui directive:

```
#include <stdio.h>
```

În final, cele două fișiere se compilează împreună inserind directive:

```
#include "fis1.cpp"
```

în fișierul *fis2.cpp* sau directive:

```
#include "fis2.cpp"
```

în fișierul *fis1.cpp*.

În acest caz fișierul *stdio.h* se va include (preprocesa) de două ori. Pentru a evita acest lucru se pot folosi directivele compilării condiționate, ca mai jos.

Se inserează la începutul fiecărui fișier secvența:

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
```

Această secvență de directive realizează următoarele:

- dacă numele `__STDIO_H` nu este definit, atunci:
  - se include fișierul *stdio.h*;
  - se definește numele `__STDIO_H`.
- în caz contrar (numele `__STDIO_H` este definit) se trece la preprocesarea textului aflat după `#endif`.

Se observă că această secvență permite includerea fișierului *stdio.h* numai în cazul în care `__STDIO_H` este nedefinit. Totodată, după includerea lui *stdio.h* se definește variabila `__STDIO_H`, ceea ce permite suprimarea includerilor ulterioare ale fișierului *stdio.h*.

Directiva `#define` de mai sus nu atribuie o valoare de substituție pentru numele `__STDIO_H` deoarece acest nume nu este folosit la substituție ci numai în directiva `#ifndef`.

Numele `__STDIO_H` a fost ales ca să fie sugestiv și pe cât posibil să nu coincidă cu unul care să aibă alte utilizări.

### Exerciții:

15.1 Să se definească un macro care apoi să fie apelat la evaluarea expresiilor de mai jos:

$$i = (|a| + |b|) / (1 + |a-b|)$$

unde:

$i, a, b$  - Sînt de tip *int*.

$$z = (|x+y| - |x-y|) / (1 + |x+y| |x-y|)$$

unde:

$x, y, z$  - Sînt de tip *double*.

Definiția macroului și apelurile lui pentru evaluarea acestor expresii intră în compunerea programului de mai jos.

Valorile variabilelor  $a, b, x$  și  $y$  se citesc de la intrarea standard.

## PROGRAMUL BXV1

```
#include <stdio.h>
#include <stdlib.h>

#define ABS(X) ((X)<0 ? -(X) : (X))

#include "BVIII2.CPP" /* pcit_int */
#include "BVIII3.CPP" /* pcit_int_lim */

int pcit_double(char *p, double *d);

main() /* citește valorile variabilelor a, b, x și y, evaluează expresiile de
      mai jos și afișează valorile variabilelor i și z:
      
$$i = (|a| + |b|) / (1 + |a - b|);$$

      
$$z = (|x + y| \cdot |x - y|) / (1 + |x + y| \cdot |x - y|).$$

*/
{
    int a, b, i;
    double x, y, z;
    double s, d;
    char er[] = "s-a tastat EOF\n";

    /* citește valoarea lui a */
    if (pcit_int_lim("a=", -32768, 32767, &a) == 0) {
        printf(er);
        exit(1);
    }

    /* citește valoarea lui b */
    if (pcit_int_lim("b=", -32768, 32767, &b) == 0) {
        printf(er);
        exit(1);
    }

    /* citește valoarea lui x */
    if (pcit_double("x=", &x) == 0) {
        printf(er);
        exit(1);
    }

    /* citește valoarea lui y */
    if (pcit_double("y=", &y) == 0) {
        printf(er);
        exit(1);
    }
}
```

```

/* calculul valorii lui i */
i = (ABS(a)+ABS(b)) / (1+ABS(a-b));

/* calculul valorii lui z */
s = ABS(x+y);
d = ABS(x-y);
z = (s-d) / (1+s*d);

/* afisare rezultate */
printf("a = %d\tb = %d\n",a,b);
printf("(ABS(a)+ABS(b))/(1+ABS(a-b))=%d\n",i);
printf("x = %f\ty = %f\n",x,y);
printf("(ABS(x+y)-ABS(x-y))/(1+ABS(x+y)*\
ABS(x-y))=%g\n",z);

} /* sfirsit main */

int pcit_double(char *p,double *d)
/* - afiseaza textul spre care pointeaza p;
   - citeste un numar si-l pastreaza in zona spre care pointeaza d;
   - returneaza zero la intalnirea sfirsitului de fisier si unu in caz
      contrar. */
{
    char t[255];
    double f;

    for(;;){
        printf("%s",p);
        if(gets(t)==0)
            return 0;
        if(sscanf(t,"%lf",&f)==1)
            break;
        printf("nu s-a tastat un numar\n");
    }
    *d=f;
    return 1;
}

```

- 15.2 Să se scrie un program în care să se definească un macro pentru calculul unui polinom de gradul doi și care să fie apelat pentru evaluarea expresiilor:

$$pol1 = 3x^2 + 7x - 8;$$

$$pol2 = x^3 - 3,5x + 1,2.$$

Programul citește valoarea lui  $x$  și afișează valorile variabilelor  $pol1$  și  $pol2$ .

## PROGRAMUL BXV2

```
#include <stdio.h>
#include <stdlib.h>

#define POL(A,B,C) ((A)*x*x+(B)*x+(C))

main() /* citește pe x, calculează și afișează valorile expresiilor:
        3x*x+7x-8
        și
        x*x-3,5x+1,2. */
{
    double x;
    char t[255];

    for(;;){
        printf("x=");
        if(gets(t)==0){
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%lf",&x)==1)
            break;
        printf("nu s-a tastat un numar\n");
    }
    printf("x = %f\t 3x*x+7x-8 = %g\n",x,POL(3,7,-8));
    printf("x = %f\t x*x-3,5x+1,2 = %g\n",x,
        POL(1,-3.5,1.2));
}
```

## 16. INTRĂRI/IEȘIRI

Limbajul C nu dispune de instrucțiuni de intrare/ieșire. Aceste operații se realizează prin intermediul unor *funcții* din biblioteca standard a limbajului. Aceste funcții pot fi aplicate în mod eficient la o gamă largă de aplicații. De asemenea, ele asigură o portabilitate bună a programelor, fiind implementate într-o formă compatibilă pe diferite sisteme de operare. Aceasta însă nu înseamnă că ele nu au și facilități suplimentare pe anumite sisteme, cum ar fi de exemplu în cazul mediului de programare Turbo C.

În acest capitol ne vom referi la funcțiile din biblioteca standard I/O care au o utilizare frecventă în diferite aplicații. Deși ele au un caracter general, în capitolul de față ne vom referi la facilitățile oferite de funcțiile existente în biblioteca compilatorului Turbo C.

Majoritatea operațiilor de intrare/ieșire se realizează în ipoteza că datele sînt organizate în *fișiere*.

În general, prin *fișier* înțelegem o colecție ordonată de elemente, numite *înregistrări*, care sînt păstrate pe diferite suporturi externe. Cele mai utilizate suporturi pentru fișiere sînt cele magnetice. Acestea, de obicei, sînt *discuri* și *benzi magnetice*. Ele se numesc suporturi *reutilizabile*, deoarece zona utilizată pentru a păstra înregistrările unui fișier poate fi ulterior reutilizată pentru a păstra înregistrările altui fișier.

Datele introduse de la tastatura unui terminal se consideră că formează un *fișier de intrare*. Înregistrarea în acest caz, de obicei, este formată din datele tastate la terminal pe un rînd deci caracterul de rînd nou (*newline*) este *terminator de înregistrare*. În mod analog, datele care se afișează pe terminal formează un *fișier de ieșire*. Înregistrarea, și în acest caz, poate fi formată din caracterele unui rînd.

Datele care se scriu la imprimantă formează și ele un fișier de ieșire. Un rînd scris la imprimantă este o înregistrare.

Un fișier are o înregistrare care marchează *sfîrșitul de fișier*. În cazul fișierelor de intrare de la tastatură sfîrșitul de fișier se generează prin secvența:

<Ctrl>-Z

De obicei, prin *intrare standard* se înțelege tastatura terminalului de la care s-a lansat programul. În mod analog, *ieșirea standard* este ecranul aceluiași terminal.

Prelucrarea fișierelor implică un număr de *operații* specifice acestora. În

primul rind, orice fișier, înainte de a fi prelucrat, trebuie *deschis*. De asemenea, la terminarea prelucrării unui fișier, acesta trebuie *închis*.

Alte operații frecvente în prelucrarea fișierelor sînt:

- crearea unui fișier;
- citirea (consultarea) înregistrărilor unui fișier;
- actualizarea unui fișier;
- adăugarea de înregistrări într-un fișier;
- poziționarea într-un fișier;
- ștergerea unui fișier.

Toate aceste operații pot fi realizate prin funcții standard existente în biblioteca limbajului C.

Tratarea fișierelor se poate face la două *nivele*. *Primul nivel* face apel direct la sistemul de operare. Acesta este *nivelul inferior* de prelucrare a fișierelor.

Celălalt nivel se realizează prin intermediul unor proceduri specializate de prelucrare a fișierelor care utilizează structuri speciale de tip FILE (fișier). Acesta este *nivelul superior* de prelucrare a fișierelor.

Pînă în prezent am utilizat funcții pentru prelucrarea la nivel superior a fișierelor standard de intrări/ieșiri la terminalul de la care s-a lansat programul.

Așa de exemplu, pentru a citi o înregistrare de la intrarea standard am utilizat funcția *gets*. Pentru a afișa înregistrarea la terminalul standard am folosit funcția *puts*. Alte funcții utilizate, care permit și conversii din format intern în cel extern și invers, sînt funcțiile *scanf* și *printf*. Prima permite citirea datelor de la intrarea standard sub controlul formatelor. Cea de a doua funcție permite afișarea standard sub controlul formatelor.

Aceste funcții au prototipurile în fișierul *stdio.h*.

Pentru citirea de caractere de la intrarea standard am folosit macroul *getchar* definit în fișierul *stdio.h*. În mod analog, pentru a afișa caractere la ieșirea standard am folosit macroul *putchar* definit tot în fișierul *stdio.h*. În continuare se indică funcții standard analoge din biblioteca C care pot fi utilizate pentru prelucrarea altor fișiere decît cele de la intrarea sau ieșirea standard.



## 16.1. Nivelul inferior de prelucrare a fișierelor

### 16.1.1. Deschiderea unui fișier

Așa cum s-a spus mai sus, orice fișier înainte de a fi prelucrat trebuie deschis. Deschiderea unui fișier existent se realizează cu ajutorul funcției *open*. La revenirea din funcția *open* se returnează așa numitul *descriptor de fișier*. Acesta este număr întreg nenegativ. El identifică în continuare fișierul respectiv în toate operațiile realizate asupra lui.

În forma cea mai simplă, funcția *open* se apelează printr-o expresie de atribuire de forma:

```
df = open(...);
```

unde:

*df* - Este o variabilă de tip *int*.

Funcția *open* are prototipul:

```
int open(const char *cale, int acces);
```

unde:

*cale* - Este un pointer spre un șir de caractere care definește calea spre fișierul care se deschide.

*acces* - Este o variabilă de tip întreg care poate avea una din valorile următoare:

- **O\_RDONLY** - fișierul se deschide numai în citire (read-only);
- **O\_WRONLY** - fișierul se deschide numai în scriere (write\_only);
- **O\_RDWR** - fișierul se deschide în citire/scriere;
- **O\_APPEND** - fișierul se deschide pentru adăugarea de înregistrări la sfârșitul lui;
- **O\_BINARY** - fișierul se prelucrează binar;
- **O\_TEXT** - fișierul este de tip text (se prelucrează pe caractere).

Aceste valori se pot combina cu ajutorul operatorului sau logic pe biți ('|'). De exemplu:

```
O_RDWR | O_BINARY
```

înseamnă că fișierul este deschis în citire/scriere binară.

În mod implicit se consideră că un fișier este de tip text, deci `O_TEXT` se poate omite.

Utilizarea funcției *open* implică includerea fișierelor *io.h* și *fcntl.h*:

```
#include <io.h>
```

și

```
#include <fcntl.h>
```

Calea spre fișier trebuie să respecte convențiile sistemului de operare MS-DOS. În cea mai simplă formă ea este un șir de caractere care definește numele fișierului, urmat de punct și extensia fișierului. Aceasta presupune că fișierul respectiv se află în directorul curent.

În cazul în care fișierul nu este în directorul curent, numele este precedat de o construcție de forma:

*litera*: \nume\_1\nume\_2\...\nume\_k

unde:

- |               |   |
|---------------|---|
| <i>litera</i> | - Definește discul (în general A, B pentru disc flexibil, C, D, ... pentru disc fix). |
| <i>nume_i</i> | - Este nume de subdirector, $i = 1, 2, \dots, k$ .                                    |

Deoarece *calea* se include între ghilimele, caracterul '\' se dublează.

Deschiderea unui fișier nu reușește dacă unul din parametri este eronat. În acest caz funcția *open* returnează valoarea -1. Un caz frecvent de eroare este acela în care se încearcă deschiderea unui fișier inexistent.

**Exemple:**

1.

```
char nfiș[] = "fiș1.dat";  
int df;  
df = open(nfiș, O_RDONLY);
```

Fișierul *fiș1.dat* din directorul curent se deschide în citire. Funcția returnează valoarea -1 dacă nu există un astfel de fișier în directorul curent.

2.

```
int d,  
d = open("A:\\JOC\\BIO.C", O_RDWR);
```

Fișierul *BIO.C* din directorul *JOC* de pe discul *A* se deschide în citire/scriere. În acest caz șirul de caractere aflat pe locul primului

parametru al apelului funcției *open* se păstrează într-o zonă de memorie specială rezervată șirurilor de caractere. Adresa de început a acestei zone de memorie se atribuie primului parametru al funcției *open* în momentul apelului.

3.

```
int i;  
i=open("c:\\borlandc\\include\\text\\text.h",O_APPEND);
```

Fișierul *text.h* se deschide pentru adăugare de înregistrări.

4.

```
int c;  
c=open("fis.c",O_WRONLY);
```

În directorul curent trebuie să existe fișierul *fis.c*. Prin deschiderea lui în scriere, acesta se crează din nou, vechiul conținut al fișierului pierzându-se.

Pentru a crea un fișier nou se utilizează funcția *creat* în locul funcției *open*.

Aceasta are prototipul:

```
int creat(const char *calea, int mod);
```

unde:

- |              |  |
|--------------|--|
| <i>calea</i> | - Este un pointer spre un șir de caractere care definește calea spre fișierul care se deschide în creare.  |
| <i>mod</i>   | - Este un întreg care poate fi definit folosind constantele simbolice de mai jos: <ul style="list-style-type: none"><li>● S_IREAD - proprietarul poate citi fișierul;</li><li>● S_IWRITE - proprietarul poate scrie în fișier;</li><li>● S_IEXE - proprietarul poate executa programul conținut în fișierul respectiv.</li></ul> |

Acești indicatori pot fi combinați folosind caracterul '|'. De exemplu, pentru citire/scriere se va folosi:

```
S_IREAD|S_IWRITE
```

Funcția *creat*, ca și funcția *open*, returnează descriptorul de fișier sau -1 în caz de eroare.

Utilizarea funcției *creat* implică includerile de fișiere:

```
#include <io.h>
```

și

```
#include <stat.h>
```

În cazul în care funcția *creat* se folosește pentru a deschide un fișier existent, atunci acesta se va șterge, urmînd ca în locul lui să se creeze fișierul nou.

Menționăm că fișierele de intrare/ieșire standard se deschid automat la lansarea programului în execuție. De aceea, aceste fișiere nu se deschid prin program de către programator.

Descriptorul de fișier reprezentat de intrarea standard are valoarea zero, iar cel reprezentat de ieșirea standard are valoarea unu.

Sistemul oferă utilizatorului încă o ieșire standard care, de obicei, este destinată pentru afișarea *erorilor*.

Fișierul corespunzător acestei ieșiri standard are descriptorul de fișier egal cu doi. Nici acest fișier nu se deschide prin program de către utilizator.

### 16.1.2. Citirea dintr-un fișier

Operația de citire a unei înregistrări dintr-un fișier deschis cu ajutorul funcției *open* se realizează folosind funcția *read*. Aceasta returnează numărul de octeți citiți din fișier sau -1 la eroare.

Prototipul funcției este:

```
int read(int df, void *buf, unsigned lung);
```

unde:

- |             |  |
|-------------|--|
| <i>df</i>   | - Este descriptorul de fișier returnat de funcția <i>open</i> la deschiderea fișierului respectiv. |
| <i>buf</i>  | - Este pointerul spre zona de memorie în care se păstrează înregistrarea citită din fișier.        |
| <i>lung</i> | - Este lungimea în octeți a înregistrării citite.  |

La fiecare apel al funcției *read* se citește înregistrarea curentă. Astfel, la primul apel se citește prima înregistrare din fișier, la al doilea apel a doua înregistrare și așa mai departe.

Ordinea înregistrărilor este cea definită la crearea fișierului și eventual la adăugarea ulterioară de înregistrări noi.

La un apel al funcției *read* se citesc cel mult *lung* octeți. La întâlnirea sfîrșitului de fișier nu se citește nimic și de aceea funcția *read* returnează zero în acest caz.

Dacă *lung=1*, atunci la apelul lui *read* se citește un singur octet din fișier. Acest lucru nu este avantajos mai ales atunci când se citesc înregistrări de pe disc. În acest caz o valoare utilizată frecvent pentru *lung* este 512.

Funcția *read* poate fi folosită pentru a citi caractere de la intrarea standard. În acest caz descriptorul de fișier are valoarea zero la apelul funcției *read*.

Utilizarea funcției *read* implică includerea fișierului *io.h*

### 16.1.3. Scrierea într-un fișier

Pentru a scrie o înregistrare într-un fișier folosim funcția *write*. Fișierul trebuie să fie în prealabil deschis cu ajutorul funcției *open* sau *creat*. Ea este asemănătoare cu funcția *read* și are același prototip. Diferența constă în aceea că realizează transferul datelor în sens invers, adică din zona de memorie spre care poartă cel de al doilea parametru al ei, în fișier.

Ea returnează numărul octeților scriși în fișier, adică o valoare care coincide cu aceea a celui de al treilea parametru al ei. În cazul în care funcția returnează o altă valoare decât valoarea celui de al treilea parametru din apel, înseamnă că scrierea a fost eronată.

Funcția *write* poate fi folosită pentru a scrie la cele două ieșiri standard, folosind descriptori de fișier de valoare 1 sau 2. Utilizarea funcției *write* implică includerea fișierului *io.h*.

### 16.1.4. Poziționarea într-un fișier

Operațiile de citire/scriere se execută *secvențial*. Aceasta înseamnă că la fiecare apel al funcției *read* sau *write* se citește înregistrarea curentă, respectiv se scrie înregistrarea în poziția curentă de pe suportul fișierului.

Înregistrările se scriu una după alta pe suportul fișierului. La citire ele se citesc în aceeași ordine în care au fost scrise la crearea fișierului. Acest mod de scriere și acces la înregistrările fișierului se numește *acces secvențial*. Accesul secvențial este util atunci când dorim să prelucrăm toate înregistrările fișierului, una după alta.

În practică apar însă situații în care noi dorim să scriem și să citim înregistrări într-o ordine diferită de cea secvențială, eventual chiar aleatoare. În acest caz se spune că *accesul* la fișier este *aleator*. Pentru a realiza un acces aleator este nevoie să ne putem poziționa oriunde în fișierul respectiv. O astfel de poziționare este posibilă pe suporturile de tip disc magnetic și se realizează folosind funcția *lseek*.

Ea are prototipul:

**long lseek (int *df*, long *deplasament*, int *origine*);**

unde:

- |                           |   |
|---------------------------|---|
| <b><i>df</i></b>          | - Este descriptorul de fișier.  |
| <b><i>deplasament</i></b> | - Definește numărul de octeți peste care se va deplasa capul de citire/scriere al discului.   |
| <b><i>origine</i></b>     | - Are una din valorile: <ul style="list-style-type: none"><li>● 0 - deplasamentul se consideră de la începutul fișierului;</li><li>● 1 - deplasamentul se consideră din poziția curentă a capului de citire/scriere;</li><li>● 2 - deplasamentul se consideră de la sfârșitul fișierului.</li></ul> |

Prin apelul funcției *lseek* nu se realizează nici un fel de transfer de informație, ci numai poziționarea în fișier. Operația următoare realizată prin apelul funcției *read* sau *write* se va realiza din această nouă poziție a capului de citire/scriere.

Funcția returnează poziția capului de citire/scriere în număr de octeți, față de începutul fișierului. În caz de eroare, se returnează -1L.

Utilizarea funcției *lseek* implică includerea fișierului *io.h*.

Apelul:

***lseek(df,0l,2)***

permite poziționarea capului de citire/scriere la sfârșitul fișierului definit de valoarea descriptorului de fișier *df*.

În mod analog, apelul:

***lseek(df,0l,0)***

permite poziționarea capului de citire/scriere la începutul fișierului definit de *df*.

### 16.1.5. Închiderea unui fișier

După terminarea prelucrării unui fișier acesta trebuie închis. Acest lucru se realizează automat dacă programul se termină prin apelul funcției *exit*.

Programatorul poate închide un fișier folosind funcția *close*. Se recomandă ca un fișier să se închidă de îndată ce s-a terminat prelucrarea lui. Aceasta din cauză că numărul fișierelor ce pot fi deschise simultan este

limitat. Această limită poate fi definită de utilizator. De obicei, ea are valoarea 20.

Menționăm că fișierele corespunzătoare intrărilor și ieșirilor standard nu se închid de către programator.

Funcția *close* are prototipul:

**int close (int df);**

unde:

*df* - Este descriptorul fișierului care se închide.

La o închidere normală a fișierului, funcția returnează valoarea zero. În caz de eroare se returnează -1.

Utilizarea funcției *close* implică includerea fișierului *io.h*.

**Exerciții:**

- 16.1 Să se scrie un program care copiază intrarea standard la ieșirea standard folosind o zonă tampon de 70 de caractere.

## PROGRAMUL BXVII

```
#include <io.h>

#define LZT 70

main() /* copiaza intrarea standard la iesirea standard */
{
    char zt[LZT];
    int lung;

    while( (lung=read(0,zt,LZT))>0)
        write(1,zt,lung);
}
```

- 16.2 Să se scrie un program care citește un șir de numere de la intrarea standard și crează două fișiere *fis1.dat* și *fis2.dat*, primul conține numerele de ordin impar citite de la intrarea standard - primul, al treilea, al cincilea etc., iar al doilea conține numerele de ordin par citite de la aceeași intrare - al doilea, al patrulea etc. Apoi se listează, la ieșirea standard, cele două fișiere în ordinea *fis1.dat*, *fis2.dat* cite un număr pe un rind în formatul:

număr de ordine număr

## PROGRAMUL BXVI2

```
#include <stdio.h>
#include <stdlib.h>
#include <sys\stat.h>
#include <fcntl.h>
#include <io.h>
#include <conio.h>

main() /* - citește un sir de numere și creează fișierele fis1.dat cu nume-
        rele de ordin impar și fis2.dat cu numerele de ordin par;
        - apoi se listează fișierele. */
{
    union unr {
        float nr;
        char tnr[sizeof(float)];
    };
    union unr nrcit;
    int df1,df2;
    int i,j;

    /* se deschid fișierele fis1.dat și fis2.dat în creare cu acces în
        citire/scriere */
    if((df1=creat("fis1.dat",S_IWRITE|S_IREAD))== -1){
        printf("nu se poate deschide fișierul\
            fis1.dat în creare\n");
        exit(1);
    }
    if((df2=creat("fis2.dat",S_IWRITE|S_IREAD))==
        -1){
        printf("nu se poate deschide fișierul\
            fis2.dat în creare\n");
        exit(1);
    }

    /* se citește sirul de numere și se păstrează în cele două fișiere */
    j=1;
    printf("tastati numerele separate prin caractere\
        albe\n");
    while ((scanf("%f",&nrcit.nr))==1){
        if(j%1){ /* este impar */
            if((write(df1,nrcit.tnr,sizeof(float)))!=
                sizeof(float)){
                printf("eroare la crearea fișierului\
```



```

        fis1.dat\n");
        exit(1);
    }
}
else /* este par */
    if((write (df2,nrcit.tnr,sizeof(float)))!=
        sizeof(float)){
        printf("eroare la crearea fisierului \
            fis2.dat\n");
        exit(1);
    }
    j++;
}

/* s-a terminat citirea numerelor si scrierea lor in fisiere */
if(close(df1)<0) {
    printf("eroare la inchiderea fisierului\
        fis1.dat\n");
    exit(1);
}
if(close(df2)<0) {
    printf("eroare la inchiderea fisierului\
        fis2.dat\n");
    exit(1);
}

/* se redeschide fisierul fis1.dat si se listeaza la iesirea standard */
if((df1=open("fis1.dat",O_RDONLY))!= -1){
    printf("nu se poate deschide fisierul\
        fis1.dat in citire\n");
    exit(1);
}
j=1;
while((i=read(df1,nrcit.tnr,sizeof(float)))>0){
    printf("%6d\t%g\n",j,nrcit.nr);
    if(j%46==0){
        printf("Actionati o tasta pentru a\
            continua\n");
        getch();
    }
    j += 2;
}
if(i<0){

```

```

        printf("eroare la citirea din fis1.dat\n");
        exit(1);
    }
    close(df1);
    printf("Actionati o tasta pentru a\
        continua \n\n\n");
    getch();

    /* se redeschide fisierul fis2.dat si se listeaza la iesirea standard */
    if((df2=open("fis2.dat",O_RDONLY))!= -1){
        printf("nu se poate deschide fisierul\
            fis2.dat in citire\n");
        exit(1);
    }
    j=2;
    while((i=read(df2,nrcit.tnr,sizeof(float)))>0){
        printf("%6d\t%g\n",j,nrcit.nr);
        if(j%46==0){
            printf("Actionati o tasta pentru a\
                continua\n");
            getch();
        }
        j += 2;
    }
    if(i<0){
        printf("eroare la citirea din fis2.dat\n");
        exit(1);
    }
    close(df2);
}

```

16.3 Să se rescrie programul din exercițiul precedent folosind un singur fișier *fis.dat*.

În acest caz se păstrează, în fișierul *fis.dat*, toate datele citite de la intrarea standard. Apoi fișierul *fis.dat* se parcurge de două ori, din două în două înregistrări, întâi începînd cu prima înregistrare, apoi începînd cu a doua.

### PROGRAMUL BXVI3

```

#include <stdio.h>
#include <stdlib.h>
#include <sys\stat.h>

```

```
#include <fcntl.h>
#include <io.h>
#include <conio.h>
```

```
main() /* - citește un sir de numere si creează fișierul fis.dat;
        - apoi listează fișierul fis.dat din două in două înregistrări;
        - întru se începe cu prima înregistrare, apoi cu a doua. */
```

```
{
    union unr {
        float nr;
        char tnr[sizeof(float)];
    };
    union unr nrcit;
    int df;
    int i,j,k;
```

```
/* se deschide fișierul fis.dat în creare cu acces în citire/scriere */
if((df=creat("fis.dat",S_IWRITE|S_IREAD))== -1){
    printf("nu se poate deschide fișierul fis.dat\
        în creare\n");
    exit(1);
}
```

```
/* se citește sirul de numere si se păstrează în fis.dat */
printf("tastați numerele separate prin caractere\
    albe\n");
while ((scanf("%f",&nrcit.nr))==1)
    if((write(df,nrcit.tnr,sizeof(float))) !=
        sizeof(float)){
        printf("eroare la crearea fișierului\
            fis.dat\n");
        exit(1);
    }
```

```
/* s-a terminat citirea numerelor si scrierea lor în fișier */
if(close(df)<0) {
    printf("eroare la închiderea fișierului\
        fis.dat\n");
    exit(1);
}
```

```
/* se redeschide fișierul fis dat si se listează la ieșirea standard */
if((df=open("fis.dat",O_RDONLY))== -1){
```

```

printf("nu se poate deschide fisierul fis.dat\
      in citire\n");
exit(1);
}
j=1;
for(k=0;k<2;k++){
    while((i=read(df,nrcit.tnr,sizeof(float)))>0){
        printf("%6d\t%g\n",j,nrcit.nr);
        if(j%46==0){
            printf("Actionati o tasta pentru a\
                  continua\n");
            getch();
        }

/* avans peste o inregistrare */
        if(lseek(df,(long)sizeof(float),1)== -1L)
            break; /* s-a ajuns la sfirsitul fisierului fis.dat */
        j += 2;
    } /* sfirsit while */
    if(i<0){
        printf("eroare la citirea din fis.dat\n");
        exit(1);
    }

/* se trece la citirea a doua: se face pozitionare pe inregistrarea a doua a
   fisierului fis.dat */
    if(k<1){
        if(lseek(df,(long)sizeof(float),0)== -1L){
            printf("nu se poate face pozitionare pe\
                  inregistrarea a doua\n");
            exit(1);
        }
        printf("Actionati o tasta pentru a\
              continua\n\n\n");
        getch();
        j=2;
    }
} /* sfirsit for */
close(df);
}

```

## 16.2. Nivelul superior de prelucrare a fișierelor

La acest nivel operațiile de prelucrare a fișierelor se execută utilizându-se funcții specializate de gestiune a fișierelor. De asemenea, fiecărui fișier i se atașează o structură de tip FILE. Acest tip este definit în fișierul *stdio.h*. De asemenea, toate funcțiile din această clasă au prototipurile în fișierul *stdio.h*.

### 16.2.1. Deschiderea unui fișier

Pentru a deschide un fișier la acest nivel de prelucrare a fișierelor se utilizează funcția *fopen*.

Ea returnează un pointer spre tipul FILE (tipul fișier) sau pointerul nul în caz de eroare.

Prototipul funcției este următorul:

**FILE \*fopen (const char \*calea, const char \*mod);**

unde:

*calea*

- Are aceeași semnificație ca și în cazul funcției *open*;

*mod*

- Este un pointer spre un șir de caractere care definește modul de prelucrare al fișierului după deschidere. Acest șir de caractere se definește astfel:

- "r" - deschidere în citire (read);
- "w" - deschidere în scriere (write);
- "a" - deschidere pentru adăugare (append);
- "r+" - deschidere pentru modificare (citire/scriere);
- "rb" - citire binară;
- "wb" - scriere binară;
- "r+b" - citire/scriere binară.

Cu ajutorul funcției *fopen* se poate deschide un fișier inexistent în modul *w* sau *a*. În acest caz, fișierul respectiv se consideră în creare.

Dacă se deschide un fișier existent în modul "w", atunci se va crea din nou fișierul respectiv și vechiul conținut al său se va pierde.

Deschiderea unui fișier în modul "a" permite adăugarea de înregistrări după ultima înregistrare existentă în fișier.

Cu ajutorul funcțiilor din această clasă se pot trata intrările și ieșirile standard.

Fișierele corespunzătoare nu se deschid de către utilizator, deoarece ele sînt deschise automat la lansarea programului.

Pentru a utiliza aceste fișiere se vor folosi următorii pointeri spre tipul FILE:

- |               |   |
|---------------|---|
| <i>stdin</i>  | - Pentru a citi de la intrarea standard.            |
| <i>stdout</i> | - Pentru a afișa pe ecranul de la ieșirea standard. |
| <i>stderr</i> | - Pentru afișarea erorilor la ieșirea standard.     |
| <i>stdprn</i> | - Ieșirea paralelă pe imprimantă.                   |
| <i>stdaux</i> | - Comunicație serială.                              |

### 16.2.2. Prelucrarea pe caractere a unui fișier

Fișierele pot fi scrise și citite caracter cu caracter folosind două funcții simple:

- |             |                   |
|-------------|-------------------|
| <i>putc</i> | - Pentru scriere. |
| <i>getc</i> | - Pentru citire.  |

Funcția *putc* are prototipul:

**int putc (int c, FILE \*pf);**

unde:

- |           |  |
|-----------|--|
| <i>c</i>  | - Este codul ASCII al caracterului care se scrie în fișier.  |
| <i>pf</i> | - Este un pointer spre tipul FILE a cărui valoare a fost returnată de funcția <i>fopen</i> la deschiderea fișierului în care se face scrierea. |
|           | - În particular, <i>pf</i> poate fi unul din pointerii:  |
|           | • <b>stdout</b> (ieșire standard);   |
|           | • <b>stderr</b> (ieșire standard pentru eroare),   |
|           | • <b>stdprn</b> (ieșire paralelă la imprimantă);   |
|           | • <b>stdaux</b> (ieșire serială).  |

Funcția *putc* returnează valoarea lui *c* sau -1 la eroare.

Macroul *putchar* se definește cu ajutorul funcției *putc* astfel:

**#define putchar(c) putc(c,stdout)**

Această definiție se află în fișierul *stdio.h*.

Funcția *getc* are prototipul:

**int** *getc* (FILE \**pf*);

unde:

*pf* - Este pointerul spre tipul FILE.

Valoarea lui *pf* este definită la apelul funcției *fopen*.

În particular, *pf* poate fi unul din pointerii:

- **stdin** (intrare standard);
- **stderr** (intrare serială).

Funcția *getc* returnează codul ASCII al caracterului citit din fișier.

Macroul *getchar* se definește în fișierul *stdio.h* astfel:

**#define** *getchar()* *getc(stdin)*

### Exerciții:

16.4 Să se scrie un program care copiază intrarea standard la ieșirea standard folosind funcțiile *getc* și *putc*.

### PROGRAMUL BXVI4

```
#include <stdio.h>

main() /* copiaza intrarea standard la iesirea standard */
{
    int c;

    while((c=getc(stdin)) != EOF)
        putc(c, stdout);
}
```

### Observație:

Programul poate fi scris și cu ajutorul macrourilor *getchar* și *putchar*. Nu avem decât să utilizăm:

*getchar()* în loc de *getc(stdin)*

și

*putchar(c)* în loc de *putc(c, stdout)*

În urma expandării acestor macrouri se obțin chiar apelurile lui *getc* și

*putc* din programul de mai sus.

### 16.2.3. Închiderea unui fișier

După terminarea prelucrării unui fișier, acesta urmează a fi închis. Închiderea unui fișier se realizează cu ajutorul funcției *fclose* de prototip:

```
int fclose (FILE *pf);
```

unde:

*pf* - Este pointerul spre tipul FILE. Valoarea lui a fost definită prin funcția *fopen* la deschiderea fișierului.

Funcția returnează valorile:

0 - La închidere normală.

-1 , - La eroare.

#### Exerciții:

16.5 Să se scrie un program care listează la imprimanta paralelă un fișier a cărui cale este argumentul din linia de comandă. În cazul în care în linia de comandă nu există un astfel de argument, se listează caracterele citite de la intrarea standard.

#### PROGRAMUL BXVI5

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
/*- listeaza la imprimanta paralela un fisier a carui cale este argumentul
   din linia de comanda;
   - daca argumentul lipseste, se listeaza caracterele de la intrarea
   standard. */
{
    FILE *pf;
    int c;

    if(argc == 1 )

/* nu exista argumente */
        pf=stdin;
    else
```



```

    if(argc==2){
/* se deschide fisierul a carui cale se afla in linia de comanda */
        if((pf=fopen(argv[1], "r"))==0){
            printf("nu se poate deschide fisierul\
                %s\n", argv[1]);
            exit(1);
        }
    }
    else{
        printf("numar argumente eronat in linia\
            de comanda\n");
        exit(1);
    }
    while((c=getc(pf))!=EOF)
        putc(c, stdout);
    fclose(pf);
}

```

## 16.2.4. Intrări/ieșiri de șiruri de caractere

Biblioteca standard a limbajului C conține funcțiile *fgets* și *fputs* care permit citirea, respectiv scrierea, înregistrărilor care sînt șiruri de caractere.

Funcția *fgets* are prototipul:

```
char *fgets (char *s, int n, FILE *pf);
```

unde:

- |           |   |
|-----------|---|
| <i>s</i>  | - Este pointerul spre zona în care se păstrează caracterele citite din fișier.          |
| <i>n</i>  | - Este dimensiunea în octeți a zonei în care se citesc caracterele din fișier.          |
|           | - La un apel a lui <i>fgets</i> se citesc cel mult <i>n-1</i> caractere.                |
| <i>pf</i> | - Este pointerul spre tipul FILE a cărui valoare s-a definit la deschiderea fișierului. |

Citirea caracterelor se întrerupe la întîlnirea caracterului '\n' sau după citirea a cel mult *n-1* caractere. În zona spre care pointează *s* se păstrează caracterul '\n' dacă acesta a fost citit din fișier, iar apoi se memorează caracterul nul ('\0').

În mod normal, funcția returnează valoarea pointerului *s*. La întîlnirea sfîrșitului de fișier funcția returnează valoarea zero.

Funcția *fputs* scrie un șir de caractere (succesiune de caractere terminată cu '\0') într-un fișier.

Ea are prototipul:

**int fputs (const char \*s, FILE \*pf);**

unde:

- s* - Este pointerul spre începutul zonei de memorie care conține șirul de caractere care se scrie în fișier.
- pf* - Este pointerul spre tipul FILE a cărui valoare a fost definită la deschiderea fișierului prin apelul lui *fopen*.

Funcția *fputs* returnează codul ASCII al ultimului caracter scris în fișier sau -1 la eroare.

Aceste două funcții sînt similare cu funcțiile *gets* și *puts* utilizate în multe din exercițiile din această carte.

## 16.2.5. Intrări/ieșiri cu format

Biblioteca standard a limbajului C conține funcții care permit realizarea operațiilor de intrare/ieșire cu format. În acest scop se pot utiliza funcțiile *fscanf* și *sprintf*.

Acestea sînt similare cu funcțiile *sscanf*, respectiv *sprintf*. Diferența dintre ele constă în faptul că *fscanf* și *sprintf* au ca prim parametru un pointer spre tipul FILE, iar *sscanf* și *sprintf* au ca prim parametru un pointer spre o zonă în care se păstrează caractere. De aici rezultă și utilizările celor două funcții. Astfel, funcția *fscanf* citește date dintr-un fișier și le convertește păstrînd rezultatele acestor conversii în conformitate cu parametri existenți la apelul funcției, în timp ce *sscanf* realizează același lucru dar utilizînd date din memorie. În mod analog, funcția *sprintf*, convertește date din format intern în format extern și apoi le scrie într-un fișier, spre deosebire de funcția *sprintf* care realizează aceleași conversii, dar rezultatele se păstrează în memorie.

Prototipul funcției *fscanf* este:

**int fscanf (FILE \*pf, const char \*format,...);**

unde:

- pf* - Este un pointer spre tipul FILE a cărui valoare a fost definită prin apelul funcției *fopen*.
- Acesta definește fișierul din care se face citirea.

Ceilalți parametri sînt identici cu cei utilizați în funcția *scanf*.

Funcția *fscanf*, ca și funcțiile *scanf* și *sscanf*, returnează numărul cîmpurilor citite corect din fișier. La întîlnirea sfîrșitului de fișier funcția returnează valoarea EOF, definită în fișierul *stdio.h*.

Funcția *fprintf* are prototipul:

**int fprintf (FILE \*pf, const char \*format,...);**

unde:

- pf*
- Este un pointer spre tipul FILE a cărui valoare a fost definită prin apelul funcției *fopen*.
  - Acesta definește fișierul în care se face scrierea.

Ceilalți parametri sînt identici cu cei utilizați în funcția *printf*.

Funcția *fprintf*, ca și funcțiile *printf* și *sprintf*, returnează numărul caracterelor scrise în fișier sau -1 în caz de eroare.

### 16.2.6. Vidarea zonei tampon a unui fișier

Există situații în care programatorul dorește să vîdeze zona tampon a unui fișier. În acest scop se poate utiliza funcția *fflush*. Ea are prototipul:

**int fflush (FILE \*pf);**

unde:

- pf*
- Este pointerul spre tipul FILE care definește fișierul pentru care se videază zona tampon.

Dacă fișierul este deschis în scriere, atunci conținutul zonei tampon se scrie în fișierul respectiv.

Dacă fișierul este deschis în citire, caracterele necitite din zona tampon se pierd, deoarece după apelul funcției zona tampon devine goală.

Un exemplu de utilizare a acestei funcții este acela al recuperării după o eroare în citire.

**Exemplu:**

```
int i,n;  
...  
  
do {  
    printf("tastati valoare lui n=");  
    if ((i=scanf("%d",&n))!=1)
```

```

        break;
    printf("nu s-a tastat un intreg\n");
    if (i==EOF) /* s-a tastat sfirsitul de fisier */
        exit(1);
    fflush(stdin); /* se elimina caracterele necitite din zona
                    tampon atasata intrarii standard */
}while(1);

```

Funcția returnează valoarea -1 în caz de eroare și zero dacă vidarea se realizează fără erori.

## 16.2.7. Poziționarea într-un fișier

Biblioteca standard a limbajului C conține funcția *fseek* care permite deplasarea capului de citire/scriere al discului în vederea prelucrării înregistrărilor fișierului într-o ordine diferită de cea secvențială.

Ea are prototipul:

```
int fseek (FILE *pf, long deplasament, int origine);
```

unde:

*pf* - Este pointerul spre tipul FILE care definește fișierul în care se face poziționarea capului de citire/scriere.

Valoarea lui se definește la deschiderea fișierului prin apelul funcției *fopen*.

Ceilalți parametri se definesc la fel ca în cazul funcției *fseek*.

Funcția *fseek* returnează valoarea zero la o poziționare corectă și o valoare diferită de zero în caz de eroare.

O altă funcție utilă în cazul accesului aleator la fișier, este funcția *ftell*. Această funcție permite să se cunoască poziția curentă a capului de citire/scriere. Ea are prototipul:

```
long ftell (FILE *pf);
```

unde:

*pf* - Este pointerul spre tipul FILE care definește fișierul în cauză.

Funcția returnează valoarea care definește poziția curentă a capului de citire/scriere.

Valoarea returnată de *ftell* este deplasamentul în octeți a poziției capului de citire/scriere față de începutul fișierului.

## 16.2.8. Prelucrarea fișierelor binare

Fișierele organizate ca date binare (octeții nu sînt considerați ca fiind coduri de caractere) pot fi prelucrate folosind funcțiile *fread* și *fwrite*.

În acest caz se consideră că înregistrarea este o colecție de *articole*. Articolul este o dată de un tip oarecare (predefinit sau definit de utilizator). La o citire, se transferă într-o zonă specială, numită *zonă tampon* (buffer), un număr de articole care se presupune că au o lungime fixă. În mod analog, la scriere se transferă din zona tampon un număr de articole de lungime fixă.

Prototipul funcției *fread* este următorul:

**unsigned fread (void \*ptr, unsigned dim, unsigned nrart, FILE \*pf);**

unde:

- ptr* - Este pointerul spre zona tampon care conține articolele citite (înregistrarea citită).
- dim* - Definește dimensiunea unui articol în octeți.
- nrart* - Definește numărul de articole din compunerea înregistrării citite.
- pf* - Este pointerul spre tipul FILE care definește fișierul din care se face citirea.

Funcția returnează numărul de articole citite sau -1 în caz de eroare.

Funcția *fwrite* are prototipul similar cu cel al funcției *fread*:

**unsigned fwrite (void \*ptr, unsigned dim, unsigned nrart, FILE \*pf);**

unde:

- ptr* - Este pointerul spre zona tampon care conține articolele care se scriu în fișier.
- dim* - Definește lungimea unui articol.
- nrart* - Definește numărul de articole din compunerea înregistrării care se scrie în fișier.
- pf* - Este pointerul spre tipul FILE care definește fișierul în care se scrie înregistrarea.

Funcția returnează numărul articolelor scrise în fișier sau -1 în caz de eroare.

Menționăm că pentru a prelucra fișiere binare, la deschiderea lor se folosește modul *b*:

**"wb"** - Pentru creare.

"rb"	- Pentru citire.
"r+b"	- Pentru citire/scriere.
"w+b"	- Idem.
"ab"	- Pentru adăugare de înregistrări.

### 16.3. Ștergerea unui fișier

Un fișier poate fi șters apelînd funcția *unlink*. Aceasta are prototipul:

```
int unlink (const char *calea);
```

unde:

*calea* - Este un pointer spre un șir de caractere identic cu cel utilizat la crearea fișierului în funcția *creat* sau *fopen*.

Funcția *unlink* returnează valoarea zero la o ștergere reușită și -1 în caz de eroare.

### 16.4. Redirecțarea fișierelor de intrare/ieșire standard

Am văzut că funcțiile: *scanf*, *printf*, *gets* și *puts*, precum și macrourile *getchar* și *putchar* citesc și scriu la terminalul standard. Ele pot fi utilizate și cu alte periferice dacă, în prealabil, se face o redirecțare a fișierelor standard de intrare/ieșire. Acest lucru se realizează folosind, în linia de comandă a apelului execuției programului, caracterele '<'și'>'.

Caracterul '<' se folosește pentru redirecțarea fișierului de intrare (*stdin*), iar caracterul '>' pentru redirecțarea fișierului de ieșire (*stdout*).

Aceste caractere se folosesc în felul următor:

```
< specificator_de_fisier_de_intrare
```

și

```
> specificator_de_fisier_de_iesire
```

Specificatorul de fișier depinde de sistemul de operare utilizat.

În cazul sistemului de operare MS-DOS specificatorul de fișier poate fi:

<i>prn</i>	- Pentru ieșirea pe imprimanta paralelă.
<i>coml</i>	- Pentru transferul de date serial.
<i>calea</i>	- Spre un fișier de pe disc.

Menționăm că fișierul de ieșire standard pentru erori (*stderr*) nu poate fi redirectat.

### Exemple:

1. Programul *BXVI4* copiază intrarea standard la ieșirea standard. Fie *BXVI4.EXE* imaginea executabilă a programului respectiv.

Un apel de forma:

```
>BXVI4.EXE >prn
```

afișează la imprimanta paralelă caracterele citite de la intrarea standard.

2. Apelul:

```
>BXVI4.EXE <fis1.dat >fis2.dat
```

crează fișierul *fis2.dat* prin copierea fișierului *fis1.dat*. Ambele fișiere sînt în directorul curent (în care se află și fișierul *BXVI4.EXE*).

3. Apelul:

```
>BXVI4.EXE <fis1.dat
```

afișează la ieșirea standard conținutul fișierului *fis1.dat*.

Menționăm că pentru a opri afișarea în vederea analizării datelor afișate, se acționează tasta *Break*. Pentru a continua afișarea se va acționa o tastă oarecare, de exemplu bara pentru spațiu.

### Exerciții:

- 16.6 Să se scrie un program care citește de la intrarea standard datele a căror format sînt definite mai jos și le scrie în fișierul *misc.dat* din directorul curent.

Fișierul este organizat binar, fiecare articol conținînd următoarele date:

tip	denumire	um	cod	preț	cantitate
I	REZISTENTA	010KO	123456789	10	10000

### PROGRAMUL BXVI6

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define MAX 50
```

```

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
} ARTICOL;
/* 6 articole in zona tampon */

union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
}buf;

int citart(ARTICOL *str)
/* - citește datele de la intrarea standard și le pastrează în zona spre care
   pointează str;
   - returnează EOF la întâlnirea sfîrșitului de fișier și 1 în caz contrar. */
{
    int c,nr;
    float x,y;

    for(;;)
        if((nr=scanf("%1s %50s %3d %2s %1d %f %f",
                     str->tip,str->den,&str->val,str->unit,
                     &str->cod,&x,&y))!=7){
            if(nr==EOF)
                return EOF;
            printf("date eronate; se reia citirea\n");
            fflush(stdin);
        }
        else
            break;
    str->pret=x;
    str->cant=y;
    return 1;
} /* sfîrșit citart */

main() /* crearea fișierului misc.dat cu datele citite de la intrarea
       standard */

```



```

{
    FILE *pf;
    int i,n;

    /* se deschide fisierul in scriere binara */
    if((pf=fopen("misc.dat","wb"))==0){
        printf("nu se poate deschide in creare\
            misc.dat\n");
        exit(1);
    }
    for(;;){

        /* se umple zona tampon a fisierului */
        for(i=0;i<6;i++){
            if((n=citart(&buf.a[i]))==EOF)
                break; /* s-a intilnit EOF */

            /* se scrie zona tampon daca nu este vida */
            if(i) /* zona tampon nu este vida */
                if(i!=fwrite(buf.zt,sizeof(ARTICOL),i,pf)){

                    /* eroare la scrierea in fisier */
                    printf("eroare la scrierea in\
                        fisier\n");
                    exit(1);
                }
            if(n==EOF)
                break;
        } /* sfirsit for */
        fclose(pf);
    } /* sfirsit main */
}

```

#### Observație:

În datele de intrare, *tip* este o literă:

- |          |                 |
|----------|-----------------|
| <i>I</i> | - Intrări.      |
| <i>E</i> | - Ieșiri.       |
| <i>T</i> | - Transfer etc. |

Tipul se citește folosind specificatorul %1s. În felul acesta se omit eventualele caractere albe care preced litera ce definește tipul.

16.7 Să se scrie un program care afișează articolele păstrate în fișierul misc.dat creat în exercițiul precedent, pentru tip=1.

#### PROGRAMUL BXVI7

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
}ARTICOL;

union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
}buf;

main() /* citește și afișează articolele din fișierul misc.dat
        pentru tip=1 */
{
    FILE *pf;
    int i,n;

    if((pf=fopen("misc.dat","rb"))==0){
        printf("nu se poate deschide fișierul\
            misc.dat\n");
        exit(1);
    }
    printf("%30s\n\n\n", "INTRARI");
    while((n=fread(buf.zt, sizeof(ARTICOL), 6, pf))>0)

    /* listează articolele dintr-o înregistrare cu tipul=1 */
        for(i=0; i<n; i++)
            if(buf.a[i].tip[0]=='I')
                printf("%-50s %03d%8s %09ld %.2f %.2f\n",
                    buf.a[i].den, buf.a[i].val,
```

```
        buf.a[i].unit,buf.a[i].cod,  
        buf.a[i].pret,buf.a[i].cant);  
    fclose(pf);  
}
```

## 17. FUNCȚII STANDARD

Mediul de programare TURBO C pune la dispoziția utilizatorului un set de funcții standard care pot fi utilizate cu succes într-o serie de aplicații din diferite domenii. Aceste funcții pot fi apelate atât din programe scrise în C cât și din cele scrise în C++. Ele pot fi grupate în mai multe clase, ținând seama de utilizările lor.

Apelurile funcțiilor standard implică includeri de fișiere de tip *h* care conțin prototipurile lor. Ca exemple de astfel de fișiere putem aminti: *stdio.h*, *stdlib.h*, *conio.h*, *io.h*, *fcntl.h*, *alloc.h* etc.

De obicei, fișierele de tip *h* relative la funcțiile standard se află în subdirectorul INCLUDE al directorului TC pentru TURBO C sau BORLANDC pentru TURBO C++.

Listarea subdirectorului INCLUDE printr-o comandă de forma:

```
dir \tc\include
```

permite afișarea numelor fișierelor de tip *h*.

În continuare se poate lista conținutul fișierului *nume.h* folosind o comandă de forma:

```
type \tc\include\nume.h
```

În cazul în care dorim să afișăm la imprimanta paralelă conținutul fișierului *nume.h*, putem utiliza comanda de mai jos:

```
type \tc\include\nume.h >prn
```

Printr-o astfel de listare se pot pune în evidență prototipurile funcțiilor standard aflate în fișierul respectiv.

Pentru a obține informații suplimentare despre o funcție standard se poate proceda ca mai jos.

1. Setare pe directorul *tc* sau *borlandc*:  

```
>cd \tc sau cd \borlandc
```
2. Se activează mediul de dezvoltare integrat TURBO C sau C++ tastind:  

```
tc sau bc
```
3. Se activează meniul de editare:  

```
<Alt> - E
```
4. Se tastează, în fereastra de editare, numele funcției standard despre

care se doresc informații.

5. Acționind tasta cu săgeată spre stînga, se vine cu cursorul pe una din literele din compunerea numelui funcției respective.

6. Se tastează

<Ctrl> - F1

Se obține o fereastră de *Help* cu explicații despre funcția respectivă. Aceste explicații conțin:

- prototipul funcției;
- fișierul sau fișierele de tip *h* care conțin prototipul funcției respective;
- descrierea pe scurt a efectului apelului funcției respective;
- nume de parametri și eventualele funcții înrudite pentru care se pot consulta informații de *help* care să vină în completarea celor de față;
- exemple de utilizare a funcției respective.

Cu ajutorul tastelor cu săgeți și a tastei TAB se pot selecta nume de parametri și de funcții înrudite în vederea obținerii de informații de *help* suplimentare. În acest scop, după selectarea unui astfel de nume se acționează tasta *Enter*. Se deschide o nouă fereastră de *help*.

În cazul în care există mai multe ferestre de *Help*, se poate trece de la o fereastră la alta folosind tasta *PgDn*.

Pentru a reveni la o fereastră anterioară vom acționa tasta *PgUp*.

Pentru a reveni dintr-o fereastră de *Help* în fereastra de editare se acționează tasta *ESC*.

Amintim că pentru a reveni în sistemul de operare se acționează:

<Alt> - X

În cele ce urmează vom trece în revistă clasele de funcții standard utilizate mai frecvent.

O primă clasă de funcții sînt cele destinate prelucrării fișierelor. Funcțiile din această clasă care se utilizează mai frecvent au fost descrise în capitolul precedent. De asemenea, funcțiile utilizate la operațiile de intrare/ieșire cu terminalul standard au fost folosite în toate capitolele cărții de față. De aceea, în acest capitol nu ne vom mai referi la funcțiile din această clasă.

O altă clasă de funcții standard se referă la alocarea dinamică a memoriei. Acestea au prototipul în fișierul *alloc.h*.

Funcțiile mai importante din această clasă au fost descrise și utilizate deja în mai multe capitole din carte. În acest capitol ne vom referi la încă o funcție din această clasă care uneori are o utilizare importantă (funcția *coreleft*).

Într-un capitol precedent au fost prezentate funcțiile mai importante utilizate la prelucrarea șirurilor: *strlen*, *strcpy*, *strcat* și *strcmp* împreună cu unele versiuni ale lor.

Ele au prototipul în fișierul *string.h*.

Aceste funcții au fost utilizate în diferite exerciții din capitolele precedente și de aceea nu ne vom mai referi la ele în capitolul de față.

O clasă importantă de funcții o constituie funcțiile de gestiune a ecranului. Aceste funcții au prototipul în fișierele *conio.h* și *graphics.h*. Ele prezintă o importanță mare în diferite aplicații și de aceea funcțiile respective vor fi descrise și aplicate în capitolele următoare.

În capitolul de față ne vom opri asupra următoarelor clase:

- funcții pentru realizarea de conversii;
- funcții de calcul;
- funcții pentru gestiunea memoriei;
- funcții de control ale proceselor;
- funcții de gestiune a datei și orei.

De asemenea, ne vom opri și asupra unor macroui definite în fișierul *ctype.h* și anume:

macroui de clasificare

și

macroui de transformare a simbolurilor.

## 17.1. Macroui de clasificare

În această clasă distingem un număr de macroui care au aplicații în prelucrarea caracterelor. Aceste macroui sînt definite în fișierul *ctype.h*.

Un prim macro din această clasă este macroul *isascii* de prototip:

```
int isascii(int c);
```

El returnează o valoare diferită de zero dacă valoarea lui *c* aparține intervalului [0,127] și zero în caz contrar.

Celelalte macroui sînt de forma:

**int is ... (int c);**

Valoarea lui *c* trebuie să aparțină intervalului [0,127]. Cele trei puncte reprezintă o succesiune de litere ca mai jos.

Aceste macrouri reprezintă o valoare diferită de zero dacă *c* este:

pentru	<b>isalpha</b>	o literă
	<b>isalnum</b>	o literă sau o cifră
	<b>isdigit</b>	o cifră
	<b>isgraph</b>	un caracter grafic
	<b>islower</b>	o literă mică
	<b>isprint</b>	un caracter imprimabil
	<b>isspace</b>	un caracter spațiu, tabulator, retur de car, interline, tabulator vertical sau salt de pagină
	<b>isupper</b>	o literă mare
	<b>isxdigit</b>	o cifră hexazecimală

## 17.2. Macrouri de transformare a caracterelor

În această clasă distingem macrouri definite tot în fișierul *cctype.h*.

Prototipurile lor sînt:

**int toascii(int c);** - returnează ultimii 7 biți ai lui *c*.

**int tolower(int c);** - transformă pe *c* din literă mare în literă mică.

**int toupper(int c);** - transformă pe *c* din literă mică în literă mare.

**Exemplu:**

Modificăm programul din exercițiul 16.4 așa încît literele mari de la intrarea standard să fie afișate ca litere mici:

```
...
while((c = getchar()) != EOF)
    if(isupper(c))
        putchar(tolower(c));
    else
        putchar(c);
...
```

## 17.3. Funcții de conversie

Biblioteca standard a limbajului C conține mai multe funcții de conversie a datelor din format extern în cel intern și invers.

Până în prezent am utilizat două astfel de funcții:

`scanf`

și

`sprintf`

Aceste funcții au prototipurile în fișierul *stdio.h*. Ele permit realizarea de conversii sub controlul formatelor.

În afară de aceste două funcții, biblioteca standard mai conține și alte funcții de conversie dintre care amintim pe cele utilizate mai frecvent.

Funcția *atoi* are prototipul:

`int atoi(const char *p);`

unde:

*p* - Pornește spre un șir de caractere în compunerea căruia intră cifre zecimale și care eventual sînt precedate de un semn.

Funcția convertește întregul zecimal definit de șirul spre care pornește *p* în binar de tip *int* și returnează rezultatul acestei conversii.

Funcția *ltoa* realizează conversia inversă.

Ea are prototipul:

`char *ltoa(int val, char *sir, int baza);`

unde:

*val* - Este valoarea binară de tip *int* care se convertește ca un întreg în baza de numerație definită de *baza*.

*sir* - Este pointerul spre zona în care se păstrează rezultatul conversiei sub formă de șir de caractere.

*baza* - Este baza rezultatului.

Funcția returnează valoarea pointerului *sir*.

Biblioteca conține funcții similare pentru tipul *long*. Astfel, funcția *atol* are prototipul:

`long atol(const char *p);`

și realizează conversia din zecimal a șirului spre care pornește *p*, în binar de tip *long*.

Conversia inversă se realizează cu ajutorul funcției *ltoa*:



**char \*ltoa(long val, char \*sir, int baza);**

Parametrii *val*, *sir* și *baza* au aceleași utilizări ca și în cazul funcției *itoa*. Funcția returnează valoarea pointerului *sir*.

Toate aceste funcții au prototipul în *stdlib.h*.

Tot din această clasă face parte și funcția *atof* de prototip:

**double atof(const char \*p);**

Această funcție convertește un număr aflat în zona spre care pointează *p*, în format flotant dublă precizie și returnează rezultatul acestei conversii. Funcția *atof* realizează aceeași conversie ca și cea realizată cu ajutorul funcției *scanf* relativ la specificatorul de format *%lf*. Aceasta înseamnă că în compunerea numărului spre care pointează *p* se poate afla caracterul punct, precum și un exponent.

Funcția *atof* are prototipul în fișierele *math.h* și *stdlib.h*.

## 17.4. Funcții de calcul

Majoritatea funcțiilor standard utilizate la calcule numerice au prototipul în fișierul *math.h*. Multe dintre acestea au prototipul:

**double nume (double x);**

unde:

*nume* - Are semnificația de mai jos:

<b>nume</b>	<b>echivalentul în limbajul matematic</b>
<b>acos</b>	<b>arccos</b>
<b>asin</b>	<b>arcsin</b>
<b>atan</b>	<b>arctg</b>
<b>cos</b>	<b>cos</b>
<b>sin</b>	<b>sin</b>
<b>exp</b>	<b>e la puterea x</b>
<b>log</b>	<b>ln</b>
<b>log10</b>	<b>log în baza 10</b>
<b>sqrt</b>	<b>rădăcina pătrată</b>
<b>pow10</b>	<b>10 la puterea x</b>
<b>ceil</b>	<b>rotunjire prin exces</b>
<b>floor</b>	<b>rotunjire prin lipsă</b>
<b>fabs</b>	<b>valoare absolută</b>

<b>sinh</b>	<b>sh</b>
<b>cosh</b>	<b>ch</b>
<b>tanh</b>	<b>th</b>

Alte funcții de calcul cu prototipurile în *math.h* și care se utilizează frecvent sînt:

**double atan2(double y, double x);**

returnează  $\arctg(y/x)$ ;

**double pow(double x, double y);**

returnează  $x$  la puterea  $y$ ;

**double cabs(struct complex z);**

returnează modulul numărului complex  $z$ ;

**double poly(double x, int n, double c[]);**

returnează valoarea polinomului în  $x$  de grad  $n$ , coeficienții fiind păstrați în tabloul  $c$  astfel:

$c[0]$  - Termen liber.

$c[1]$  - Coeficientul lui  $x$ .

...  
 $c[i]$  - Coeficientul lui  $x$  la puterea  $i$ .

...  
 $c[n]$  - Coeficientul lui  $x$  la puterea  $n$ .

Următoarele funcții de calculul au prototipurile în fișierul *stdlib.h*:

**int abs(int x);**

returnează valoarea absolută a lui  $x$  de tip *int*;

**long labs(long x);**

returnează valoarea absolută a lui  $x$  de tip *long*;

**int rand(void);**

returnează un număr natural pseudo-aleator mai mic decît 32768;

**int random(int nr);**

returnează un număr natural pseudo-aleator mai mic decît  $nr$ ;

**void srand(int n);**

setează sămînța numerelor pseudo-aleatoare la valoare lui  $n$ .

O parte din funcțiile de calcul amintite mai sus au fost utilizate în diferite exerciții din capitolele acestei cărți.

## 17.5. Funcții pentru gestiunea memoriei

Funcțiile din această clasă au prototipurile în fișierul *alloc.h*.

În paragraful 8.8 au fost amintite funcțiile *malloc*, *calloc*, *free*, și versiunile acestora pentru cazul în care pointerii sunt de tip *far* : *farmalloc*, *farcalloc* și *farfree*.

În acest paragraf amintim funcția *coreleft* care are prototipul:

**unsigned coreleft (void);**

Această funcție se utilizează pentru modelele de memorie: *tiny*, *small* și *medium* (pentru aceste modele pointerii sunt de tip *near*). Ea returnează dimensiunea memoriei libere în momentul apelului ei.

Pentru modelele de memorie *compact*, *large* și *huge* (pointerii sunt de tip *far*) funcția *coreleft* are prototipul:

**unsigned long coreleft(void);**

Menționăm că în cazul mediilor integrate de dezvoltare Turbo C și C++, modelul se definește cu ajutorul submeniului *Compile* al meniului *Options*.

## 17.6. Funcții de control ale proceselor

Indicăm mai jos prototipurile unor funcții mai importante din această clasă.

Funcția *exit* are prototipul:

**void exit(int stare);**

Ea termină execuția programului.

Parametrul *stare* are valoarea zero la o terminare normală a execuției programului și diferită de zero pentru o terminare anormală. Funcția *exit* vedează zonele tampon ale fișierelor deschise în scriere, în momentul apelului ei. Apoi se închid toate fișierele deschise.

Funcția *system* are prototipul:

**int system(const char \*comanda);**

unde:

*comanda* - Este un pointer spre un șir de caractere care reprezintă o comandă MS-DOS. Ea are ca efect execuția comenzii definite de șirul de caractere spre care pointează parametrul *comanda*.

Funcția returnează codul de stare al execuției comenzii:

0 - La terminare normală.  
-1 - La eroare.

Funcția *spawnl* este asemănătoare cu funcția *system*. Ea permite lansarea în execuție a imaginii executabile a unui program păstrat într-un fișier.

Prototipul funcției este:

`int spawnl(int mod, char *cale, char *arg0, char *arg1, ..., NULL);`

unde:

*mod* - Are ca valoare constantă simbolică `P_WAIT`.  
- Această valoare înseamnă că programul din care se apelează funcția *spawnl* își întrerupe execuția până la terminarea execuției programului lansat prin apelul funcției *spawnl*.  
*cale* - Este pointerul spre șirul de caractere care definește fișierul cu imaginea executabilă a programului care se lansează.  
*arg0, arg1, ...* - Sunt pointeri spre șiruri de caractere care reprezintă argumentele programului care se lansează.

După o execuție cu succes a programului apelat, funcția *spawnl* returnează valoarea codului de stare de la apelul funcției *exit* prin care s-a terminat execuția programului respectiv.

În caz de eroare (nu se reușește lansarea în execuție a unui program), funcția *spawnl* returnează valoarea -1.

Aceste funcții au prototipul în fișierele *stdlib.h* și *process.h*.

## 17.7. Funcții pentru gestiunea datei și a orei

Indicăm mai jos prototipurile a patru funcții pentru citirea/setarea datei și a orei. Ele implică includerea fișierului *dos.h*.

Funcțiile pentru citirea datei și orei au prototipurile:

`void getdate(struct date *d);`

și

```
void gettime(struct time *t);
```

Funcțiile pentru setarea datei și orei au prototipurile:

```
void setdate(struct date *d);
```

```
void settime(struct time *t);
```

Structurile *date* și *time* sînt definite în *dos.h* astfel:

```
struct date {  
    int da_year;  
    int da_day;  
    int da_mon;  
};  
struct time {  
    unsigned char ti_min;  
    unsigned char ti_hour;  
    unsigned char ti_hund;  
    unsigned char ti_sec;  
};
```

## 17.8. Diferite funcții de uz general

Funcția *clrscr* de prototip:

```
void clrscr(void);
```

șterge fereastra activă sau tot ecranul dacă n-a fost activat în prealabil o fereastră (definirea și activarea unei ferestre se va descrie într-un capitol următor; în mod implicit *clrscr* șterge tot ecranul).

Prototipul funcției se află în fișierul *conio.h*.

Funcția *kbhit* de prototip:

```
int kbhit(void);
```

returnează o valoare diferită de zero dacă există un caracter disponibil de la tastatură; în caz contrar ea returnează valoarea zero.

Funcția *kbhit* se poate apela ciclic pentru a aștepta tastarea unui caracter la terminalul standard. Astfel, ciclul:

```
while(!kbhit())  
    ;
```

se execută pînă în momentul în care se acționează o tastă la terminalul

standard.

Pe perioada execuției ciclului se afișează ecranul utilizator și de aceea acest ciclu se poate utiliza la fel ca și apelul funcției *getch* pentru a afișa ecranul respectiv.

Funcțiile care urmează și încep cu prefixul *str* implică includerea fișierului *string.h*.

Funcția *strlwr* are prototipul:

**char \*strlwr(char \*sir);**

Ea convertește literele mari din zona spre care pointează *sir* în litere mici.

Funcția returnează valoarea *sir*.

Funcția *strupr* de prototip:

**char \*strupr(char \*sir);**

este inversa funcției *strlwr*.

Funcția *strset* de prototip:

**char \*strset(char \*sir, int c);**

inițializează zona spre care pointează *sir* cu valoarea lui *c* (ultimii 8 biți, cei mai puțini semnificativi). Substituirea valorii lui *c* se realizează până la întâlnirea caracterului NUL, care rămâne nemodificat. Funcția returnează valoarea lui *sir*.

O variantă a acestei funcții este funcția *strnset*. Ea are prototipul:

**char \*strnset(char \*sir, int c, unsigned n);**

Funcția are aceeași acțiune ca și funcția *strset*, cu deosebirea că se inițializează cu valoare lui *c* numai cel mult primele *n* caractere ale zonei spre care pointează *sir*.

Funcția *strchr* are prototipul:

**char \*strchr(const char \*sir, char c);**

Se caută prima apariție a caracterului *c* în zona spre care pointează *sir*.

Funcția returnează pointerul spre poziția determinată sau zero în cazul în care nu există o intrare a caracterului *c* în zona respectivă.

Funcția *strrchr* are prototipul:

**char \*strrchr(const char \*sir, char c);**

Ea are aceeași acțiune ca și funcția *snchr*, cu deosebirea că determină nu prima apariție a caracterului *c*, ci ultima.

Funcțiile care urmează au prototipul în fișierul *dos.h*.

Funcția *delay* are prototipul:

**void delay(unsigned i);**

Ea suspendă execuția programului pentru o perioadă de *i* milisecunde.

Funcția *sleep* are prototipul:

**void sleep(unsigned i);**

Ea suspendă execuția programului pentru o perioadă de *i* secunde.

Funcția *sound* are prototipul:

**void sound(unsigned h);**

Ea pornește difuzorul calculatorului cu un ton egal cu *h* Hz.

Funcția *nosound* are prototipul:

**void nosound(void);**

Ea oprește sunetul de la difuzorul calculatorului.

## 17.9. Tratarea erorilor

Distingem două mari categorii de erori:

erori care apar la compilare;

și

erori care apar la execuție.

Erorile de compilare sînt de trei nivele:

- |                     |   |
|---------------------|---|
| <i>erori fatale</i> | <ul style="list-style-type: none"><li>- Semnifică o eroare în compilator.</li><li>- Dacă aceste erori nu provin de la apelul incorect al unei definiții de macro, atunci ele vor fi semnalate firmei Borland.</li></ul> |
| <i>erori</i>        | <ul style="list-style-type: none"><li>- Compilatorul afișează, în fereastra de mesaje, pentru fiecare eroare, un text explicit al-acesteia.</li><li>- Sînt prevăzute aproximativ 140 de cazuri de eroare.</li></ul>     |
| <i>avertismente</i> | <ul style="list-style-type: none"><li>- Sînt mesaje care indică:<ul style="list-style-type: none"><li>● Situații care în anumite cazuri pot constitui erori.</li></ul></li></ul>  |

- Informații cu privire la portabilitatea și calitatea programului. Sînt prevăzute circa 30 de astfel de cazuri.

Erorile care apar la execuție pot fi tratate prin:

- Valorile returnate de DOS prin intermediul variabilelor globale *errno*, *doserrno*, *sys\_errlist*, *sys\_nerr*. Aceste variabile sînt declarate în fișierele *<errno.h>* și *<dos.h>*. Funcția *perror* afișează mesajul de eroare corespunzător.
- Gestiunea erorilor prin funcția *harderr* din *<dos.h>*.
- Detecția erorilor prin funcția *ferror* în prelucrarea fișierelor.
- Gestiunea erorilor de calcul prin apelul funcției *matherr*.

### Exerciții:

17.1 Programul de mai jos afișează data și ora furnizate de sistemul de operare. Formatul de afișare este:

zz/ll/aaaa ora hh:mm:ss

### PROGRAMUL EX711

```
#include <stdio.h>
#include <dos.h>

main() /* afiseaza data si ora */
{
    struct date d;
    struct time t;

    getdate(&d);
    gettime(&t);
    printf("\n\t%02d/%02d/%04d", d.da_day, d.da_mon,
        d.da_year);
    printf("\tora\
    %02d:%02d:%02d\n", t.ti_hour, t.ti_min, t.ti_sec);
}
```

17.2 Să se scrie un program care afișează un șir de numere naturale pseudo-aleatoare mai mici decît  $10 \cdot (\text{sec} + 1)$ , unde prin *sec* s-a notat secunda din ora curentă.



## PROGRAMUL BXVII2

```
#include <dos.h>
#include <stdlib.h>
#include <stdio.h>

main() /* - afiseaza 3 numere pseudo-aleatoare mai mici decit
        10*(sec+1);
        - sec este secunda curenta. */
{
    struct time ora_crt;
    unsigned sec;
    int i;

    gettime(&ora_crt);
    sec=ora_crt.ti_sec;
    sec++;
    for(i=0;i<3;i++)
        printf("%d\n",random(10*sec));
}
```

- 17.3 Să se scrie un program care afișează un șir de numere naturale pseudo-aleatoare mai mici decît  $n$ , citit de la intrarea standard. Primul număr se afișează automat, iar celelalte în urma acționării unei taste.

Sămînța șirului de numere pseudo-aleatoare  $s$ , se calculează astfel:

$$s=(ti\_min*60 + ti\_hour*3600 + ti\_sec) \% 65535;$$

Programul se întrerupe la tastarea cifrei zero.

## PROGRAMUL BXVII3

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <conio.h>

main () /* afiseaza numere pseudo-aleatoare din intervalul [0,n] */
{
    struct time ora_crt;
    int i,n,s;

    for(;;){
        printf("n=");
```

```

    if((i=scanf("%d",&n))==1&& n>0)
        break;
    printf("nu s-a tastat un intreg pozitiv\n");
    if(i==EOF){
        printf("s-a tastat EOF \n");
        exit(1);
    }

    fflush(stdin); /* videaza zona tampon a fisierului de
                    intrare standard */
}
gettime(&ora_crt);
s=(3600L*ora_crt.ti_hour+
    60*ora_crt.ti_min+ora_crt.ti_sec)%65535;
srand(s);
for(;;){
    printf("%d\n",random(n));
    printf("Pentru a continua actionati o tasta\
        diferita de zero\n");
    if(getch()=='0')
        exit(0);
}
}

```

### Observație:

Numerele pseudo-aleatoare obținute în acest fel se pot utiliza în diferite aplicații pentru a simula situații reale. Dacă  $n=6$ , atunci numerele pseudo-aleatoare rezultate, mărite cu 1, pot simula aruncarea unui zar.

17.4 Să se scrie un program care citește numere de la intrarea standard și le copiază într-un fișier binar pe disc. Calea spre fișier este definită de argumentul programului. În caz de eroare, programul se termină apelînd funcția *exit* cu următoarele coduri de stare:

- |   |  |
|---|--|
| 1 | - Nu se poate deschide fișierul definit de argumentul programului. |
| 2 | - Eroare la scrierea numerelor în fișier.                          |
| 3 | - Eroare la închiderea fișierului.                                 |
| 4 | - Fișier vid.  |

La terminarea programului fără erori se apelează funcția *exit* cu codul de stare zero.

## PROGRAMUL BXVIII4

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
/* citește numere de la intrarea standard și le păstrează în fișierul a cărui
   cale este definită de argumentul argv[1] */
{
    FILE *pf;
    int i, n;
    double f[10];
    void *zt;
    double x;

    if(argc!=2){
        printf("nr arg=ad\n" ,argc);
        printf("%s:numar argumente eronat\n",argv[0]);
        exit(1);
    }
    if((pf=fopen(argv[1],"wb"))==0){
        printf("%s:nu se poate deschide\
              fișierul:%s\n",
              argv[0],argv[1]);
        exit(1);
    }
    n=0;
    zt=(void *)f;
    printf("Tastati numerele separate prin caractere\
           albe\n");
    for(;;){
        for(i=0;i<10;i++){
            if(scanf("%lf",&x)!=1)
                break;
            f[i]=x;
        }
        if(i==0)
            break;
        if(fwrite(zt,sizeof(double),i,pf)!=i){
            printf("%s:eroare la scrierea în\
                  fișierul:%s\n",
                  argv[0],argv[1]);
            exit(2);
        }
    }
}
```

```

        n++;
        if(i<10)
            break;
    }
    if(n==0){
        printf("%s:nu s-a citit nici un\
            numar\n",argv[0]);
        exit(4);
    }
    if(fclose(pf)==0)
        exit(0);
    printf("%s:eroare la inchiderea fisierului:%s\n",
        argv[0],argv[1]);
    exit(3);
}

```

17.5 Să se scrie un program care realizează următoarele:

- Citește numere de la intrarea standard și le păstrează în fișierul *dnr.dat*.
- Folosind utilitarul *pkzip.exe*, determină forma condensată *dnr.zip* a fișierului *dnr.dat*.
- Șterge fișierul *dnr.dat*.

Punctul *a* se realizează apelînd programul din exercițiul precedent. În acest scop vom folosi funcția *spawnl*.

Punctul *b* se realizează apelînd utilitarul *pkzip* folosind funcția *system*.

La punctul *c* se apelează funcția *unlink*.

Utilitarul *pkzip.exe* se află în directorul *arc* de pe discul *c*. Se va utiliza linia de comandă:

```
c:\arc\pkzip.exe -a dnr.zip dnr.dat
```

## PROGRAMUL BXVIIIS

```

#include <stdio.h>
#include <process.h>

main() /* citește numere de la intrarea standard si le pastreaza
        în fisierul dnr.zip */
{
    char *text[]={
        "",

```

```

"cale spre fisier eronata\n",
"eroare la scriere in fisier\n",
"nu se poate inchide fisierul\n",
"fisier vid\n"
);
int cod;

if((cod=
spawnl(P_WAIT,"C:\\borlandc\\dest\\BXVII4.EXE",
      "", "dnr.dat", 0))!=0){
    printf("\n\n****%s\n",text[cod]);
    exit(1);
}
unlink("dnr.zip");
if(system("c:\\arc\\pkzip.exe -a\
      dnr.zip dnr.dat")!=0){
    printf("eroare la apelul utilitarului\
      pkzip\n");
    exit(1);
}
if (unlink("dnr.dat")!=0){
    printf("eroare la stergerea fisierului\
      dnr.dat\n");
    exit(1);
}
}
}

```

#### Observație:

Utilitarul *pkzip* afișează o serie de informații la terminalul standard care uneori poate afecta aspectul ecranului cu datele aplicației. Pentru a elimina acest inconvenient, se poate face o redirectare pentru a afișa datele respective pe un alt periferic. În acest caz funcția *system* se va apela folosind șirul de caractere:

```
"c:\\arc\\pkzip.exe -a dnr.zip dnr.dat >nume"
```

În acest caz, informațiile afișate de *pkzip* se vor scrie în fișierul *nume* din directorul curent. Aceste informații pot fi șterse dacă redirectarea se face spre perifericul NUL, adică înlocuind *nume* din șirul de caractere de mai sus cu NUL.

17.6 Să se scrie un program care citește de la intrarea standard un text și-l redă prin sunet utilizând alfabetul Morse.

Se redau numai literele din compunerea cuvintelor. Restul caracterelor se neglijează.

Pentru redarea *punctului* se utilizează un semnal cu o frecvență de 1000 de Hz pe o durată de 0,075 secunde.

Pentru redarea *liniei* se utilizează același semnal, dar pe o durată de trei ori mai mare.

De asemenea, între două caractere din compunerea unei litere se face pauză de 0,075 secunde. Între două litere din compunerea aceluiași cuvânt se face o pauză de 0,225 secunde, iar între două cuvinte o pauză de 0,375 secunde.

O codificare similară a unui text este propusă în exercițiul P.131 din *Gazeta de Informatică* nr. 7 - 8/1992 (pag 48). În acest caz textul se codifică printr-o secvență de cifre 0 și 1.

#### PROGRAMUL DXY.C

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>

#include "bx48.cpp"

#define MAX 1000
#define FRECV 1000
#define UNIT 75
#define INTERCAR 225
#define INTERCUV 375
```

```
main() /* citește un text de la intrarea standard și-l reda prin sunet
        utilizând alfabetul Morse */
```

```
{
    char *pcuv[MAX];
    int c,i,n;
    static char *morse[]={
        ".-","-...","-.-.-","-.-.", ".","-.-.-",
        "--.", "...", ".","-.-.-", "-.-.", "-.-.",
        "--.", "-.", "--.", "-.-.", "--.", "-.-.",
        "...", "-.", "-.-.", "-.-.", "-.-.", "-.-.-", "-.-.-",
        "--.-"
    };
};
```

```

char *p,*q;

/* se citesc cuvintele din text si se pastreaza in memoria heap */
for(n=0;n<MAX;n++){
    if((pcuv[n]=citcuv())==0)
        break; /* s-a intilnit EOF */
    if(n==0){
        printf("nu s-a citit nici un cuvint\n");
        exit(1);
    }
}

/* redarea textului prin sunet conform alfabetului Morse */
for(i=0;i<n;i++){
    p=pcuv[i];
    while(*p){
        if((c=*p++)>='a')
            c=c-'a'; /* litera mica */
        else
            c=c-'A'; /* litera mare */
        for(q=morse[c];*q;){
            sound(FRECV);
            if(*q=='.'){
                delay(UNIT); /* punct */
            }
            else
                delay(3*UNIT); /* linie */
            nosound();
            if(++q) /* nu s-a terminat litera curenta */
                delay(UNIT);
        }
        if(*p) /* s-a terminat o litera */
            delay(INTERCAR);
    }
}
/* s-a terminat un cuvint */
delay(INTERCUV);
}
}

```

## 18. GESTIUNEA ECRANULUI ÎN MOD TEXT

Biblioteca standard a limbajului C și C++ conține funcții pentru gestiunea ecranului. Acesta poate fi gestionat în două moduri:

mod text;



mod grafic.

În capitolul de față prezentăm funcțiile standard mai importante utilizate la gestiunea ecranului în mod text.

În capitolul următor se tratează gestiunea ecranului în mod grafic.

Toate funcțiile standard de gestiune a ecranului în mod text au prototipurile în fișierul *conio.h*.

Modul *text* presupune că ecranul este format dintr-un număr de linii și un număr de coloane. În mod curent se utilizează 25 de linii a 80 sau 40 de coloane fiecare. Aceasta înseamnă că ecranul are o capacitate de  $25 \cdot 80 = 2000$  sau  $25 \cdot 40 = 1000$  caractere.

Poziția pe ecran a unui caracter se definește printr-un sistem de două coordonate întregi:

(*x*,*y*)

unde:

- x* - Este numărul coloanei în care este situat caracterul.
- y* - Este numărul liniei în care este situat caracterul.

Colțul din stînga sus al ecranului are coordonatele (1,1). Colțul din dreapta jos al ecranului are coordonatele (80,25) sau (40,25).

În mod implicit, funcțiile de gestiune a ecranului în mod text au acces la tot ecranul. Accesul poate fi limitat la o parte din ecran utilizînd așa numitele *ferestre*. *Fereastra* este un dreptunghi care este o parte a ecranului și care poate fi gestionată independent de restul ecranului.

Un caracter de pe ecran, pe lângă coordonate, mai are și următoarele atribute:

- culoarea caracterului afișat;
- culoarea fondului;



- clipirea caracterului.

Aceste atribute sînt dependente de adaptorul grafic utilizat. Cele mai utilizate adaptoare sînt:

- placa MDA, care este un adaptor monocrom;
- placa Hercules, care este un adaptor monocrom;
- placa CGA, care este un adaptor color;
- placa EGA, care este un adaptor color;
- placa VGA, care este un adaptor color de mare performanță.

Pentru adaptoarele color de mai sus, se pot utiliza 8 culori de fond și 16 pentru afișarea caracterelor.

Atributul unui caracter se definește cu ajutorul formulei:

$$(1) \text{ atribut} = 16 * \text{culoare\_fond} + \text{culoare\_caracter} + \text{clipire}.$$

unde:

<i>culoare_fond</i> (background)	- Este o cifră din intervalul [0,7] și are semnificația din tabela de mai jos.
<i>culoare_caracter</i> (foreground)	- Este un întreg din intervalul [0,15] și are semnificația din tabela de mai jos.
<i>clipire</i>	- Are valoarea 128 (clipirea caracterului) sau 0 (fără clipire).

În tabelul de mai jos se indică corespondența dintre valorile numerice și culorile definite de ele cu ajutorul relației (1).

Culoare	Constantă simbolică	Valoare
negru	BLACK	0
albastru	BLUE	1
verde	GREEN	2
turcoaz	CYAN	3
roșu	RED	4
purpuriu	MAGENTA	5
maro	BROWN	6
gri deschis	LIGHTGRAY	7
gri închis	DARKGRAY	8
albastru deschis	LIGHTBLUE	9
verde deschis	LIGHTGREEN	10
turcoaz deschis	LIGHTCYAN	11
roșu deschis	LIGHTRED	12

Culoare	Constantă simbolică	Valoare
purpuriu deschis	LIGHTMAGENTA	13
galben	YELLOW	14
alb	WHITE	15
clipire	BLINK	128

În paragrafele următoare se indică funcțiile standard mai importante pentru gestiunea ecranului în mod text.

## 18.1. Setarea ecranului în mod text

Se realizează cu ajutorul funcției *textmode*. Aceasta are prototipul:

```
void textmode(int modtext);
```

unde:

*modtext* - Poate fi exprimat numeric sau simbolic în felul următor:

Modul text activat	Constantă simbolică	Valoare
Caractere albe pe fond negru; 40 de coloane	BW40	0
Color 40 de coloane	C40	1
Caractere albe pe fond negru; 80 de coloane	BW80	2
Color 80 de coloane	C80	3
Monocrom 80 de coloane	MONO	7
Color cu 43 linii pentru placa EGA și 50 de linii pentru placa VGA	C4350	64
Modul precedent	LASTMODE	-1

Modul MONO se poate seta pe un adaptor monocolor.

Celelalte moduri se pot seta pe adaptoare color.

## 18.2. Definirea unei ferestre

După setarea ecranului în mod text, acesta are caracteristicile indicate în paragraful precedent.

Adesea dorim să partajăm ecranul în zone care să poată fi gestionate independent. Aceasta se realizează cu ajutorul ferestrelor.

O *ferastră* este o zonă dreptunghiulară de pe ecran. Ea se poate defini cu ajutorul funcției *window*. Prototipul ei este:

```
void window(int stnga,int sus, int dreapta,int jos);
```

unde:

(*stnga, sus*) - Reprezintă coordonatele colțului din stnga sus al ferestrei.

(*dreapta,jos*) - Reprezintă coordonatele colțului dreapta jos al ferestrei.

La un moment dat o singură fereastră este *activă* și anume aceea definită de ultimul apel al funcției *window*.

Funcțiile de gestiune a ecranului în mod text acționează totdeauna asupra ferestrei active.

După setarea modului text cu ajutorul funcției *textmode*, este activ tot ecranul.

Menționăm că funcția *window* nu are nici un efect dacă parametri de la apel sînt eronați.

## 18.3. Ștergerea unei ferestre

Fereastră activă se șterge cu ajutorul funcției *clrscr*. Ea are prototipul:

```
void clrscr(void);
```

După apelul funcției *clrscr*, fereastră activă (sau tot ecranul, dacă nu s-a definit în prealabil o fereastră prin apelul funcției *window*) devine vidă. Fondul ei are culoarea definită prin culoarea de fond (*background*) curentă.

Funcția *clrscr* poziționează *cursorul* pe caracterul din stnga sus al ferestrei active, adică în poziția de coordonate (1,1) a ferestrei active.

## 18.4. Gestiunea cursorului

Utilizatorul poate plasa cursorul pe un caracter al ferestrei folosind

funcția *gotoxy*. Ea are prototipul:

```
void gotoxy(int coloana,int linie);
```

unde:

(coloana,linie) - Reprezintă coordonatele caracterului pe care se plasează cursorul; aceste coordonate sint relative la fereastra activă.

Dacă coordonatele de la apel sint în afara ferestrei active, atunci apelul funcției este ignorat.

Poziția cursorului din fereastra activă se poate determina cu ajutorul a două funcții, care au prototipurile:

```
int wherex(void);
```

returnează numărul coloanei în care se află cursorul;

```
int wherey(void);
```

returnează numărul liniei în care se află cursorul.

Există cazuri cînd se dorește *ascunderea* cursorului. Acest lucru se poate realiza printr-o secvență specială în care se utilizează funcția *geninterrupt*. O secvență de acest fel este următoarea:

```
void ascundecursor() /* face invizibil cursorul */
{
    _AH = 1;
    _CH = 0x20;
    geninterrupt(0x10);
}
```

Cursorul poate fi reafîșat apelînd funcția de mai jos:

```
void afiscursor() /* face vizibil cursorul */
{
    _AH = 1;
    _CH = 6;
    _CL = 7;
    geninterrupt(0x10);
}
```

Amintim că *\_AH*, *\_CH* și *\_CL* sint nume utilizate pentru regiștrii calculatorului.

## 18.5. Determinarea parametrilor ecranului

Utilizatorul are posibilitatea să obțină parametri curenți ai ecranului prin apelarea funcției *gettextinfo*. Ea are prototipul:

```
void gettextinfo(struct text_info *p);
```

unde structura *text\_info* este definită în fișierul *conio.h* astfel:

```
struct text_info {  
    unsigned char winleft;  
    unsigned char wintop;  
    unsigned char winright;  
    unsigned char winbottom;  
    unsigned char attribute;  
    unsigned char normattr;  
    unsigned char currmode;  
    unsigned char screenheight;  
    unsigned char screenwidth;  
    unsigned char currx;  
    unsigned char curry;  
};
```

După apelul funcției *gettextinfo* structura de tip *text\_info*, spre care poartă *p*, este completată cu următoarele informații:

- amplasarea colțurilor ferestrei;
- culoarea fondului, a caracterelor și clipirea acestora;
- modul curent;
- dimensiunea ecranului;
- poziția cursorului.

## 18.6. Modurile video alb/negru

Modurile video alb/negru sînt două:

modul intens

și

modul normal.

*Modul intens* se obține apelînd funcția *highvideo* de prototip:

```
void highvideo(void);
```

*Modul normal* se obține cu ajutorul funcției *lowvideo* de prototip:

**void lowvideo(void);**

Intensitatea inițială este de obicei cea normală. Se poate reveni la intensitatea normală dacă se apelează funcția *normvideo* de prototip:

**void normvideo(void);**

## 18.7. Setarea culorilor

Culoarea fondului se setează cu ajutorul funcției *textbackground* de prototip:

**void textbackground(int culoare);**

unde:

*culoare* - Este un întreg în intervalul [0,7] și are semnificația definită în tabelul de la începutul acestui capitol.

Culoarea caracterelor se setează cu ajutorul funcției *textcolor* de prototip:

**void textcolor(int culoare);**

unde:

*culoare* - Este un întreg din intervalul [0,15] și are semnificația definită în tabelul de la începutul acestui capitol.

Se pot seta ambele culori, precum și clipirea caracterului folosind funcția *textattr* de prototip:

**void textattr(int atribut);**

unde:

*atribut* - Se definește cu ajutorul relației (1).

### Exerciții:

18.1 Să se scrie o funcție care afișează parametrii ecranului.

### FUNCȚIA BXVIII1

```
void pocr() /* afiseaza parametrii ecranului */
{
    struct text_info parecr;
    clrscr();
}
```

```

gettextinfo(&parecr);
printf("stinga: %u sus: %u dreapta: %u jos: %u\n",
       parecr.winleft, parecr.wintop,
       parecr.winright, parecr.winbottom);
printf("atribut: %u mod curent: %u\n",
       parecr.attribute, parecr.currmode);
printf("inaltimea ecranului: %u latimea\
       ecranului: %u\n", parecr.screenheight,
       parecr.screenwidth);
printf("coloana cursorului: %u linia\
       cursorului: %u\n", parecr.curx, parecr.cury);
}

```

- 18.2 Să se scrie un program care setează pe rând modurile text, definite cu ajutorul constantelor simbolice:

BW40, C40, BW80, C80 și C4350

și afișează parametrii ecranului pentru fiecare din modurile respective.

#### PROGRAMUL 18.VIII

```

#include <stdio.h>
#include <conio.h>

#include "bxviii1.cpp"

main() /* seteaza modurile definite cu ajutorul constantelor simbolice
       BW40, C40, BW80, C80 si C4350 si afiseaza parametrii
       ecranului in fiecare caz */
{
    int i;
    int tab[] = {BW40, C40, BW80, C80, C4350};
    char *text[] = {
        "BW40", "C40", "BW80", "C80", "C4350"
    };

    for(i=0; i<5; i++){
        textmode(tab[i]);
        pscr();
        printf("\n\t\t\t\t%s\n\n", text[i]);
        printf("Actionati o tasta pentru a\
               continua\n");
        getch();
    }
}

```

```
}
}
```

#### Observație:

Programul de față presupune prezența la calculator a unui adaptor color de tip EGA sau VEGA.

## 18.8. Gestiunea textelor

Pentru afișarea caracterelor colorate în conformitate cu atributele definite prin relația:

$$\text{atribut} = 16 * \text{culoare\_fond} + \text{culoare\_caracter} + \text{clipire}$$

se pot folosi funcțiile:

- putch* - Afișează un caracter.
- cputs* - Afișează un șir de caractere (este analogă cu funcția *puts*).
- cprintf* - Afișează date sub controlul formatelor de conversie (este analogă cu funcția *printf*).

Toate aceste funcții au prototipul în fișierul *conio.h*.

Atributul de culoare și clipire se setează cu ajutorul funcțiilor indicate în paragraful 18.7.

Biblioteca standard a limbajului C conține și alte funcții utile în gestiunea textelor. Dintre acestea amintim pe cele mai importante.

Operațiile de ștergere și inserare de linie se pot realiza prin funcțiile de mai jos:

```
void insline(void);
```

inserează o linie cu spații în fereastră; liniile de sub poziția curentă a cursorului se deplasează în jos cu o poziție,

```
void clreol(void);
```

șterge sfârșitul liniei începând cu poziția cursorului,

```
void delline(void);
```

șterge toată linia pe care este poziționat cursorul.

Un text poate fi copiat dintr-o zonă dreptunghiulară a ecranului în alta, folosind funcția *movetext*. Ea are prototipul:



**int movetext(int stanga, int sus, int dreapta, int jos,  
int stanga\_dest, int sus\_dest);**

unde:

- |  |  |
|--|--|
| <i>stanga,sus</i>                      | - Definesc începutul textului care se copiază.   |
| <i>dreapta,jos</i>                     | - Definesc sfârșitul textului care se copiază.   |
| <i>stanga_dest,</i><br><i>sus_dest</i> | - Definesc poziția primului caracter al textului după<br>copiere. Celelalte poziții rezultă automat din structura<br>textului. |

**Funcția returnează valoarea:**

- |          |                                  |
|----------|----------------------------------|
| <i>1</i> | - Dacă textul s-a copiat corect. |
| <i>0</i> | - La eroare.                     |

Textele dintr-o zonă dreptunghiulară pot fi salvate într-o zonă de memorie sau citite dintr-o astfel de zonă folosind funcțiile *gettext* și *puttext*. Ele au prototipurile de mai jos:

**int gettext(int stanga, int sus, int dreapta, int jos, void \*destinatie);**

unde:

- |   |   |
|---|---|
| <i>stanga,sus</i><br><i>dreapta,jos</i> | - Definesc o zonă dreptunghiulară din ecran care conține<br>textul de salvat. |
| <i>destinatie</i>                       | - Este pointerul spre zona de memorie în care se salvează<br>textul.          |

**Funcția returnează:**

- |          |                         |
|----------|-------------------------|
| <i>1</i> | - La copiere cu succes. |
| <i>0</i> | - La eroare.            |

**int puttext(int stanga, int sus, int dreapta, int jos, void \*sursa);**

unde:

- |   |  |
|---|--|
| <i>stanga,sus</i><br><i>dreapta,jos</i> | - Definesc o zonă dreptunghiulară din ecran în care se va<br>afișa textul citit din memorie. |
| <i>sursa</i>                            | - Este pointerul spre zona de memorie din care se<br>transferă textul pe ecran.              |

**Funcția returnează:**

- |          |                         |
|----------|-------------------------|
| <i>1</i> | - La copiere cu succes; |
| <i>0</i> | - La eroare.            |

Menționăm că fiecare caracter de pe ecran se păstrează pe doi octeți:

- pe un octet caracterul;
- pe octetul următor atributul caracterului.

#### Exerciții:

- 18.3 Să se scrie un program care afișează texte în modurile video intens și video normal.

#### PROGRAMUL BXVIII3

```
#include <conio.h>

main() /* afiseaza texte in modurile video intens si normal */
{
    textmode(BW80);
    window(10,4,60,4);
    clrscr();
    lowvideo();
    cputs("lowvideo");
    highvideo();
    cputs(" highvideo");
    normvideo();
    cputs(" normvideo");
    textmode(LASTMODE);
    cprintf("\n\nActionati o tasta pentru a\
        continua" );
    getch();
}
```

- 18.4 Să se scrie un program care afișează toate combinațiile de culori posibile pentru fond și caractere. Se consideră că se dispune de un adaptor color EGA/VGA.

#### PROGRAMUL BXVIII4

```
#include <conio.h>
#include <stdio.h>

main() /* - afiseaza toate combinatiile de culori posibile pentru
        fond si caractere;
        - se dispune de un adaptor EGA/VGA. */
{
    static char *tculoare[] = {
        "    0    BLACK                negru",
```

```

"    1    BLUE          albastru",
"    2    GREEN        verde",
"    3    CYAN         turcoaz",
"    4    RED          rosu",
"    5    MAGENTA      purpuriu",
"    6    BROWN        maro",
"    7    LIGHTGRAY    gri deschis",
"    8    DARKGRAY     gri inchis",
"    9    LIGHTBLUE    albastru deschis",
"   10    LIGHTGREEN    verde deschis",
"   11    LIGHTCYAN     turcoaz deschis",
"   12    LIGHTRED      rosu deschis",
"   13    LIGHTMAGENTA  purpuriu deschis",
"   14    YELLOW        galben",
"   15    WHITE         alb"
};

int i,j,k;
struct text_info atr;

gettextinfo(&atr);
for(i=0; i < 8; i++){

/* i alege culoarea fondului */
    window(3,2,60,20);
    k=2;
    textbackground(i);
    clrscr();
    for(j=0; j < 16; j++, k++) {
        textcolor(j);
        gotoxy(2,k);

/* j alege culoarea caracterului */
        if(i == j)
            continue;
        cputs(tculoare[j]);
    }
    gotoxy(1,18);
    printf("actionati o tasta pentru a\
continua\n");
    getch();
}
window(atr.winleft,atr.wintop,

```

```

        atr.winright, atr.winbottom);
    textattr(atr.attribute);
    clrscr();
}

```

18.5 Să se scrie un program pentru rezolvarea problemei turnurilor din Hanoi.

Această problemă a fost rezolvată în exercițiul 9.6. Mai jos se rezolvă aceeași problemă prin imagini pentru  $n < 7$ ,  $n$  fiind numărul de discuri ( $n > 0$ ). Se presupune că se dispune de un adaptor color EGA/VGA.

Programul folosește o matrice de ordinul  $3 \times 6$  pentru a păstra discurile pe cele trei tije. Numim *stiva* tabloul care păstrează elementele acestei matrice. Ea este de tip *int*. Amintim că discul de cel mai mic diametru se numerează cu 1, discul cu diametru imediat mai mare decît acesta se numerează cu 2 și așa mai departe. Dacă sînt  $n$  discuri, atunci discul cu cel mai mare diametru se numerează cu  $n$ .

Discurile aflate pe tija A se păstrează ca elemente ale tabloului *stiva* pentru care primul indice este zero:

*stiva*[0][0], *stiva*[0][1], *stiva*[0][2], ...

Discurile aflate pe tija B se păstrează ca elemente ale aceluiași tablou pentru care primul indice are valoarea unu:

*stiva*[1][0], *stiva*[1][1], *stiva*[1][2], ...

În mod analog, pentru discurile aflate pe tija C se utilizează elementele pentru care primul indice are valoarea doi:

*stiva*[2][0], *stiva*[2][1], *stiva*[2][2], ...

Inițial discurile sînt pe tija A, deci

*stiva*[0][ $i$ ] =  $n-i$  pentru  $i = 0, 1, \dots, n-1$ .

Un alt tablou utilizat în program determină numărul discurilor aflate pe fiecare tijă. Numim *istiva* acest tablou. El are 3 elemente: *istiva*[0] are ca valoare numărul discurilor aflate pe tija A, *istiva*[1] are ca valoare numărul discurilor de pe tija B, iar *istiva*[2] are ca valoare numărul discurilor de pe tija C.

Inițial *istiva*[0] =  $n$ , *istiva*[1] = 0 și *istiva*[2] = 0 deoarece toate cele  $n$  discuri se află pe tija A.

Cele trei tije se reprezintă prin trei dreptunghiuri albastre. Primul are coordonatele:

- colțul din stînga sus (7,3);
- colțul din dreapta jos (8,17).

Următorul se obține făcînd o translație cu 28 de coloane spre dreapta; deci acesta are coordonatele (7+28,3) și (8+28,17).

Ultimul dreptunghi se obține printr-o nouă translație tot de 28 de coloane.

Sub fiecare dreptunghi se afișează cu galben litera tijei pe care o reprezintă (A, B sau C).

Discurile se reprezintă prin dreptunghiuri colorate și acestea au culorile:

discul	n	LIGHTGRAY
	n-1	BROWN
	n-2	MAGENTA
	n-3	RED
	n-4	CYAN
	n-5	GREEN

Fondul ecranului este negru.

Mutările discurilor se realizează cu ajutorul funcției recursive *hanoi*. Aceasta este analogă cu cea definită în exercițiul 9.6. În cazul de față, pentru  $n < 7$ , în loc să se afișeze mutarea sub forma unui text, se realizează deplasarea prin imagini a discului care se mută de pe o tijă pe alta. Acest lucru se realizează apelînd funcția *transfer*. Aceasta are prototipul:

```
void transfer(int tija1, int tija2);
```

Ea transferă discul din vrful tijei *nja1*, în vrful tijei *nja2*.

## PROGRAMUL BXVIII5

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
```

```
int stiva[3][6];
int istiva[3];
int m;
```

```
typedef struct {
    int x;
    int y;
```

```

int u;
int v;
int poz;
}F;

struct text_info parecr;

void inithanoi (int n) /* initializeaza discurile pe tija A */
{
    int x,y,u,v,i,lit,culoare;

    /* salveaza parametrii ecranului */
    gettextinfo (&parecr);

    /* fond negru pentru tot ecranul */
    textbackground (BLACK);

    /* culoarea caracterelor se seteaza la galben */
    textcolor (YELLOW);

    /* sterge ecranul */
    clrscr ();

    /* se amplaseaza cele trei tije de culoare albastra fiecare */
    x = 7;
    y = 3;
    u = 8;
    v = 17;
    lit= 'A';
    for (i = 0; i<3; i++){
        window(x,y,u,v);
        textbackground(BLUE);
        clrscr();
    }

    /* fereastra pentru litera */
    window (x,v,u,v+2);

    /* fond negru si caracter galben = 14 */
    textattr (14);
    clrscr ();

    /* scrie litera */
    putch (lit);
}

```

```

/* se modifica parametrii pentru tija urmatoare */
    lit++;
    x = x+28;
    u = u+28;
}

/* se pun discurile pe tija A */
/* culorile discurilor au valorile 7,6,...,2 */
/* discul n are lungimea 2^n+1 caractere si inaltimea de 1 caracter */
culoare = 7;
x = 7-n;
y = 15;
u = 8+n;
for (i = n; i>0; i--){
    window (x,y,u,v-1);
    textbackground (culoare);
    clrscr ();
    culoare--;
    x++;
    u--;
    v = y;
    y = y-2;
}

/* se initializeaza tablourile stiva si istiva */
for (i = 0; i<n; i++)
    stiva [0][i] = n-i;
istiva[0] = 1;
istiva[1] = 0;
istiva[2] = 0;
getch(); /* afiseaza starea initiala */
} /* sfirsit inithanoi */

void fereastr (F *frstr,int nrstv,
               int vs,int ndisc)
/* defineste fereastr active pe discul ndisc aflat in virful vs pe tija
   nrstv */
{
    int poz;

    /* se determina pozitia tijei */
    poz = (nrstv*4 + 1)*7; /* nrstv = 0 tija A;
                             1 tija B;

```

2 tija C;  
poz - pozitia tijei. \*/

```
/* se determina coordonatele discului ndisc */
frstr -> x = poz - ndisc;
frstr -> u = poz+1+ndisc;
frstr -> y = 17 -2*vs; /* vs - numar discuri pe tija nrstv */
frstr -> v = frstr -> y +1;
```

```
/* se activeaza fereastra pe discul ndisc */
window (frstr -> x, frstr -> y, frstr -> u,
        frstr -> v);
frstr -> poz = poz; /* pastreaza pozitia tijei */
} /* sfirsit fereastra */
```

```
void transfer ( int tija1,int tija2)
/* deplaseaza discul din virful tijei tija1 in virful tijei tija2 */
{
    int i,j,k,l,nr;
    F cf;
```

```
/* determina numarul tijei tija1 */
/* tija A are numarul 0, tija B are numarul 1, tija C are numarul 2 */
if (tija1 == 'A')
    i = 0;
else
    if (tija1 == 'B')
        i = 1;
    else
        i = 2;
```

```
/* i - numarul tijei tija1;
   j - numarul discurilor pe tija tija1;
   nr - numarul discului din virful tijei tija1. */
j = istiva[i];
nr = stiva[i][j-1];
```

```
/* - determina fereastra cu discul din virful tijei a i-a;
   - aceasta devine activa. */
fereastra (&cf,i,j,nr);
```

```
/* sterge fereastra din virful stivei i */
textbackground (BLACK);
```



```

clrscr ();

/* reface tija a i-a */
window ( cf.poz, cf.y, cf.poz+1, cf.v);
textbackground (BLUE);
clrscr ();

/* - se scoate discul din stiva a i-a;
   - acesta corespunde discului care a fost in virful tijei tija1 */
istiva[i]--;

/* determina numarul tijei tija2 */
if (tija2 == 'A')
    i=0;
else
    if (tija2 == 'B')
        i = 1;
    else
        i = 2;

/* pune discul nr pe tija a i-a */
stiva[i] [istiva[i]] = nr;
istiva[i]++;

/* determina fereastra pe tija a i-a pentru discul nr */
fereasta (&cf,i,istiva[i],nr);

/* fereasta activa devine fereasta de pe tija tija2 in virful careia se va
   pune discul nr */
/* stabileste culoarea ferestrei */
textbackground(7-m+nr);
clrscr ();
} /* sfirsit transfer */

void hanoi(int n,int a,int b,int c)
/* functia recursiva pentru rezolvarea problemei turnurilor din Hanoi */
{
    if (n == 1)
        if (m>6){
            printf("se muta discul 1 de pe tija: %c pe\
                   tija: %c\n",a,b);
            getch();
            return;
        }
}

```

```

    }
    else{
        transfer(a,b);
        getch();
        return;
    }
    hanoi(n-1,a,c,b);
    if(m > 6){
        printf("discul: %d se muta de pe tija :%c pe\
              tija: %c\n",n,a,b);
        getch();
    }
    else{
        transfer (a,b);
        getch();
    }
    hanoi(n-1,c,b,a);
} /* sfirsit hanoi */

main () /* rezolva problema turnurilor din Hanoi */
{
    int i,c;
    char t[255];

    for(;;){
        printf("numarul discurilor=");
        if(gets(t)==0){
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(t,"%d",&m)==1&&m>0)
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
    }
    if(m < 7){
        printf("rezolvare insotita de imagini\n");
        inithanoi(m);
    }
    else
        printf("rezolvarea nu este insotita de\
              imagini\n");
    hanoi (m,'A','B','C');
    if(m < 7){

```

```

        window (1,1,parecr.screenwidth,
                parecr.screenheight);
        textattr (parecr.attribute);
        clrscr();
    }
} /* sfirsit main */

```

18.6 Să se scrie o funcție care afișează o fereastră limitată de un chenar și pe fondul căreia se afișează un întreg. Cursorul devine invizibil la afișarea ferestrei.

Funcția are prototipul:

```

void fereastra(int st, int sus, int dr, int jos, int fond,
               int culoare, int chenar, int n);

```

unde:

- |                 |  |
|-----------------|--|
| <i>(st,sus)</i> | - Coordonatele colțului din stnga sus.   |
| <i>(dr,jos)</i> | - Coordonatele colțului din dreapta jos.   |
| <i>fond</i>     | - Întreg din intervalul [0,7] care definește culoarea de fond a ferestrei.   |
| <i>culoare</i>  | - Întreg din intervalul [0,15] care definește culoarea pentru afișarea caracterelor.   |
| <i>chenar</i>   | - Definește tipul chenarului: <ul style="list-style-type: none"> <li>● 0 - fără bordură;</li> <li>● 1 - linie simplă;</li> <li>● 2 - linie dublă;</li> <li>● n - numărul întreg care se afișează în fereastră începînd cu punctul de coordonate relative (3,3).</li> </ul> |

Zona de ecran în care se afișează fereastra se păstrează în memoria *heap* înainte de a se afișa fereastra. Adresa acestei zone de memorie se pune pe o stivă definită cu ajutorul unui tablou de pointeri spre tipul *void*. Acest tablou este global și are 100 de elemente. El se definește astfel:

```

void far *stiva[100];

```

Locul liber în tablou se definește cu ajutorul variabilei globale *istiva*:

```

int istiva;

```

## FUNCȚIA BXVIII6

```

void orizontal(int,int);

```

```

void vertical(int,int,int,int);

void fereastră(int st,int sus,int dr,int jos,
               int fond,int culoare,int chenar,int n)

/* afiseaza o fereastră limitată de un chenar și pe fondul careia se afiseaza
numarul ferestrei */
{
    extern ELEM far *stiva[];
    extern int istiva;

/* memoreaza partea din ecran pe care se va afisa fereastră */
    if(istiva==MAX){
        printf("\nprea multe ferestre\n");
        exit(1);
    }
    if((stiva[istiva]=
        (ELEM *)farmalloc(sizeof(ELEM)))==0){
        printf("memorie insuficientă\n");
        exit(1);
    }
    if((stiva[istiva]->zoner=
        farmalloc(2*(dr-st+1)*(jos-sus+1)))==0){
        printf("\nmemorie insuficientă\n");
        exit(1);
    }
    stiva[istiva]->x=st;
    stiva[istiva]->y=sus;
    stiva[istiva]->u=dr;
    stiva[istiva]->v=jos;
    if((gettext(st,sus,dr,jos,stiva[istiva]->
        zoner))==0){
        printf("\neroare la memorarea ecranului\n");
        exit(1);
    }
    istiva++;

/* activeaza fereastră și o afiseaza pe ecran */
    window(st,sus,dr,jos);
    textattr(16*fond+culoare);
    clrscr();

/* trasare chenar */

```

```

if(chenar){
    textcolor(WHITE);
    highvideo();

/* coltul stinga sus */
    switch(chenar){
        case SIMPLU:
            putch(218);
            break;
        case DUBLU:
            putch(201);
            break;
    }

/* chenar orizontal sus */
    orizontal(dr-st-2,chenar);

/* coltul dreapta sus */
    switch(chenar){
        case SIMPLU:
            putch(191);
            break;
        case DUBLU:
            putch(187);
            break;
    }

/* chenar vertical stinga */
    vertical(jos-sus,1,2,chenar);

/* coltul stinga jos */
    gotoxy(1,jos-sus+1);
    switch(chenar){
        case SIMPLU:
            putch(192);
            break;
        case DUBLU:
            putch(200);
            break;
    }

/* chenar orizontal jos */
    orizontal(dr-st-2,chenar);

```

```

/* chenar vertical dreapta */
vertical(jos-sus-1,dr-st,2,chenar);

/* coltul dreapta jos */
gotoxy(dr-st,jos-sus+1);
switch(chenar){
    case SIMPLU:
        putch(217);
        break;
    case DUBLU:
        putch(188);
        break;
}
normvideo();
textattr(16*fond+culoare);
} /* sfirsit afisare chenar */

/* scrie pe n in fereastra */
gotoxy(3,3);
cprintf("%d",n);

/* ascunde cursor */
_AH=1;
_CH=0x20;
geninterrupt(0x10);

} /* sfirsit fereastra */

void orizontal(int a,int chenar)
/* traseaza un chenar orizontal */
{
    while(a--)
        switch(chenar){
            case SIMPLU:
                putch(196);
                break;
            case DUBLU:
                putch(205);
                break;
        }
}

```

```

void vertical(int a,int col,int lin,int chenar)
{
    while(a--) {
        gotoxy(col,lin++);
        switch(chenar){
            case SIMPLU:
                putch(179);
                break;
            case DUBLU:
                putch(186);
                break;
        }
    }
}

```

- 18.7 Să se scrie un program care afișează ferestre pe ecran în mod aleator. Ferestrele sînt de dimensiune fixă, dar au poziții aleatoare pe ecran. De asemenea, ele pot avea chenar format dintr-o linie simplă sau dublă sau să nu aibă chenar. Culorile de fond și de afișare a caracterelor sînt aleatoare. Ferestrele se numerează și numărul ferestrei se afișează în fereastră. Prima fereastră afișată se numerează cu 1.

După afișarea unei ferestre se va acționa o tastă oarecare, corespunzătoare codului ASCII, pentru a afișa fereastra următoare. Se revine la fereastra precedentă dacă se acționează tasta ESC.

Execuția programului se termină în cazul în care se acționează tasta ESC în momentul în care nu este afișată nici o fereastră.

Se presupune că se dispune de un adaptor color EGA/VGA.

## PROGRAMUL BXVIII7

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>

#define MAX 100
#define ESC 0x1b
#define SIMPLU 1
#define DUBLU 2

```

```

typedef struct {
    int x,y,u,v;
    void far *zonfer;
} ELEM;
ELEM far *stiva[MAX];

int istiva;

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bxviii6.cpp" /* fereastra */

main() /* afiseaza ferestre pe ecran in mod aleator */
{
    int c,culoare,fond,i,inalt,j,lung,s,stanga,sus;
    struct text_info info,crt;
    struct time ora_crt;

    istiva=0;
    clrscr();

    /* pastreaza tot ecranul */
    if((stiva[istiva]=
        (ELEM *)farmalloc(sizeof(ELEM)))==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    if((stiva[istiva]->zonfer=farmalloc(2*80*25))==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    if(gettext(1,1,80,25,stiva[istiva]->zonfer)==0){
        printf("nu se poate salva ecranul\n");
        exit(1);
    }
    istiva++;

    /* salveaza parametrii ecranului */
    gettextinfo(&info);

    /* citeste dimensiunile ferestrelor: lungimea si inaltimea */
    if(pcit_int_lim("lungime:",8,70,&lung)==0){
        printf("s-a tastat EOF\n");
    }

```



```

        exit(1);
    }
    if(pcit_int_lim("inaltime:",5,15,&inalt)==0){
        printf("s-a tastat EOF\n");
        exit(1);
    }

/* coordonatele maxime pentru coltul din stinga sus a ferestrelor */
i=79-lung;
j=25-inalt;

/* seteaza saminta pentru sirul de numere pseudo-aleatoare care
   definesc parametrii ferestrelor:
   - coltul din stinga sus;
   - atributul de culoare. */
gettime(&ora_crt);
s=(3600L*ora_crt.ti_hour+60*ora_crt.ti_min+
    ora_crt.ti_sec)%65535;
srand(s);
printf("Actionati o tasta pentru a continua\n");
printf("cu ESC se revine la ecranul precedent\n");
for(;;){
    c=getch();
    if(c!=ESC){

/* s-a tastat un caracter diferit de ESC */
/* se genereaza parametrii ferestrei */
        stanga=random(i)+1;
        sus=random(j)+1;
        fond=random(8);
        while((culoare=random(15)+1)==fond)
            ;
        fereastra(stanga,sus,stanga+lung,sus+inalt,
            fond,culoare,istiva%3,istiva);
        continue;
    }

/* s-a tastat ESC */
    if(--istiva>0){

/* se reface zona din ecran eliminind fereastra activa */
        puttext(stiva[istiva]->x,stiva[istiva]->y,
            stiva[istiva]->u,stiva[istiva]->v,

```

```

        stiva[istiva]->zonfer);
        farfree(stiva[istiva]);
    }
    else

/* se intrerupe executia programului */
        break;

    }
    puttext(info.winleft, info.wintop, info.winright,
            info.winbottom, stiva[0]);
    window(1,1,80,25);
    farfree(stiva[0]);
    textattr(info.attribute);

/* afiseaza cursorul */
    _AH=1;
    _CH=6;
    _CL=7;
    geninterrupt(0x10);
    clrscr();
}

```

# 19. GESTIUNEA ECRANULUI ÎN MOD GRAFIC

Modul *grafic* presupune că ecranul este format din "puncte luminoase" (*pixeli*). Numărul acestora depinde de adaptorul grafic și se numește *rezoluție*. O rezoluție este cu atât mai bună cu cât este mai mare.

Adaptorul CGA are o rezoluție de 200 de rânduri a 640 de coloane, iar adaptorul EGA oferă o rezoluție de tot atâtea coloane, dar de 350 de rânduri.

Adaptorul VGA oferă o rezoluție de 480 de rânduri a 640 de coloane sau chiar mai mare, de până la 768 de rânduri, a 1024 de coloane.

Pentru gestiunea ecranului în mod grafic se pot utiliza peste 60 de funcții standard aflate în biblioteca sistemului. Aceste funcții au prototipul în fișierul *graphics.h*.

În acest capitol indicăm funcțiile mai importante care permit gestiunea ecranului în mod grafic.

Aceste funcții folosesc pointeri de tip *far* (32 de biți).

## 19.1. Setarea modului grafic

Modul grafic se setează cu ajutorul funcției *initgraph*. Această funcție poate fi folosită singură sau împreună cu o altă funcție numită *detectgraph* care determină parametrii adaptorului grafic. Prototipul ei este:

```
void far detectgraph(int far *gd, int far *gm);
```

unde:

- În zona spre care poartea *gd* se păstrează una din valorile:

Constantă simbolică	Valoare
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8

VGA	9
PC3270	10

- În zona spre care pointează *gm* se memorează una din valorile:

*Pentru CGA:*

Constantă simbolică	Valoare
CGAC0	0
CGAC1	1
CGAC2	2
CGAC3	3

Toate aceste valori corespund unei rezoluții de 320\*200 puncte și permit maximum 4 culori.

CGAHI	4
-------	---

Permite o rezoluție de 640\*200 puncte și lucrează numai în alb/negru.

*Pentru EGA:*

Constantă simbolică	Valoare
EGALO	0

Are o rezoluție de 640\*200 puncte și permite maximum 16 culori.

EGAHI	1
-------	---

Are o rezoluție de 640\*350 puncte.

*Pentru VGA:*

Constantă simbolică	Valoare
---------------------	---------

VGALO	0
-------	---

Are o rezoluție de 640\*200 puncte.

VGAMED	1
--------	---

Are o rezoluție de 640\*350 puncte.

VGAHI	2
-------	---

Are o rezoluție de 640\*480 puncte.

Valorile spre care pointează *gd* definesc niște funcții standard corespunzătoare adaptorului grafic. Aceste funcții se numesc *drivere*. Ele se află în subdirectorul BGI.

Funcția *detectgraph* detectează adaptorul grafic prezent la calculator și păstrează valoarea corespunzătoare acestuia în zona spre care pointează *gd*.

Modul grafic se definește în așa fel încît el să fie cel mai performant pentru adaptorul grafic curent.

Cele mai utilizate adaptoare sînt cele de tip EGA și VGA. De aceea, în cele ce urmează, vom presupune că adaptorul curent este de tip EGA. În felul acesta, toate exercițiile din acest capitol pot fi rulate la un calculator echipat cu un adaptor EGA sau VGA.

Apelul funcției *detectgraph* trebuie să fie urmat de apelul funcției *initgraph*. Aceasta setează modul grafic în conformitate cu parametri stabiliți de apelul prealabil al funcției *detectgraph*.

Funcția *initgraph* are prototipul:

```
void far initgraph(int far *gd, int far *gm, int far *cale);
```

unde:

- |                        |   |
|------------------------|---|
| <i>gd</i> și <i>gm</i> | - Sînt pointeri care au aceeași semnificație ca în cazul funcției <i>detectgraph</i> .  |
| <i>cale</i>            | - Este pointer spre șirul de caractere care definește calea subdirectorului BGI care conține driverele. De exemplu, dacă BGI este subdirector al directorului BORLANDC, atunci vom folosi șirul de caractere: |

```
"c:\\BORLANDC\\BGI"
```

### Exemplu:

Pentru setarea în mod implicit a modului grafic, putem utiliza secvența de mai jos:

```
int driver, mod_grafic;  
...  
detectgraph(&driver, &mod_grafic);  
initgraph(&driver, &mod_grafic,  
          "c:\\BORLANDC\\BGI");
```

După apelul funcției *initgraph* se pot utiliza celelalte funcții standard de gestiune grafică a ecranului.

Din modul grafic se poate ieși apelînd funcția *closegraph* de prototip:

```
void far closegraph(void);
```

Funcția *initgraph* poate fi apelată folosind secvența de mai jos:

```
int driver, mod_grafic;  
...  
driver = DETECT;  
initgraph(&driver, &mod_grafic,  
          "c:\\BORLANDC\\BGI");
```

Constanta simbolică *DETECT* este definită în fișierul *graphics.h* alături de celelalte constante simbolice care definesc driverul. Aceasta are valoarea zero.

Prin apelul de mai sus, funcția *initgraph* apelează funcția *detectgraph* pentru a defini parametri implicați ai adaptorului grafic.

Utilizatorul poate defini el însuși parametri pentru inițializarea modului grafic. De exemplu, secvența:

```
int driver = 1; /* CGA */  
int mod_graf = 0; /* CGA00 */  
initgraph(&driver, &mod_graf, "c:\\BORLANDC\\BGI");
```

setează modul grafic corespunzător unui adaptor grafic CGA cu rezoluția 320\*200 puncte.

În afara acestor funcții, utilizatorul mai poate utiliza și funcția *setgraphmode* care selectează un mod grafic diferit de cel activat implicit prin *initgraph*. Această funcție are prototipul:

```
void far setgraphmode(int mode);
```

unde:

<i>mode</i>	- Are valorile:
	0 - 4      pentru CGA;
	0 - 1      pentru EGA;
	0 - 2      pentru VGA.

Ea poate fi utilizată împreună cu funcția *restorecrtmode* de prototip:

```
void far restorecrtmode(void);
```

Această funcție permite revenirea la modul precedent, iar *setgraphmode* realizează trecerea inversă.

Alte funcții din această categorie sînt:

```
void far graphdefaults(void);
```

repune parametri grafici la valorile implicite;

```
int far getgraphmode(void);
```

returnează codul modului grafic;

```
char *far getmodename(int mod);
```

returnează pointerul spre numele modului grafic definit de codul numeric *mod*;

```
char *far getdrivername(void);
```

returnează pointerul spre numele driverului corespunzător adaptorului curent;

```
void far getmoderange(int grafdriv, int far *min, int far *max);
```

definește valorile minimale și maxime ale modului grafic utilizat;

*grafdriv* are ca valoare una din valorile 1-10 indicate mai sus la funcția *detectgraph*.

Valoarea minimă a modului grafic este păstrată în zona spre care pointează *min*, iar cea maximă în zona spre care pointează *max*.

### Exerciții:

19.1 Să se scrie un program care setează modul grafic în două feluri:

- cu ajutorul funcției *detectgraph*;
- fără această funcție.

Programul afișează rezultatele setării.

## PROGRAMUL BXIX1

```
#include <conio.h>
#include <stdio.h>
#include <graphics.h>
```

```
main() /*seteaza modul grafic si afiseaza parametrii setarii */
```

```
{
    int gdriv,gmod;
    int mod,min,max;
```

```
/* setare prin apelul functiei detectgraph */
    detectgraph(&gdriv,&gmod);
    printf("valori dupa apelul functiei\
        detectgraph\n");
```

```

printf("driver=%d\tmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentu a continua\n");
getch();
initgraph(&gdriv,&gmod,"c:\\borlandc\\bgi");
printf("valori dupa initgraph\n");
printf("driver=%d\tmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();

/* setare fara apelul lui detectgraph */
gdriv=DETECT;
initgraph(&gdriv,&gmod,"c:\\borlandc\\bgi");
printf("initializare fara detectgraph\n");
printf("driver=%d\tmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();

/* afiseaza numele adaptorului grafic curent */
printf("adaptor grafic: %s\n",getdrivername());
mod=getgraphmode();
printf("cod mod=%d\tmod grafic:%s\n",mod,
      getmodename(mod));
getmoderange(gdriv,&min,&max);
printf("domeniul pentru adaptorul\
      grafic=[%d,%d]\n",min,max);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();
}

```

## 19.2. Gestiunea culorilor

Adaptoarele grafice sint prevăzute cu o zonă de memorie în care se păstrează date specifice gestiunii ecranului. Această zonă de memorie poartă denumirea de *memorie video*.

În mod grafic, ecranul se consideră format din puncte luminoase numite *pixel*. Poziția pe ecran a unui pixel se definește printr-un sistem binar:

(x,y)

unde:

x                      - Definește coloana în care este afișat pixelul.



y

- Definește linia în care este afișat pixelul

În cazul adaptoarelor color, unui pixel îi corespunde o culoare.

Culoarea pixelilor se păstrează pe biți în memoria video. Memoria video necesară pentru a păstra starea ecranului setat în mod grafic, se numește *pagină video*. Adaptoarele pot conține mai multe pagini video.

Gestiunea culorilor este dependentă de tipul de adaptor grafic existent la microprocesor.

În cele ce urmează vom avea în vedere adaptoarele grafice de tip EGA/VGA.

În mod concret, ne vom referi la facilitățile oferite de adaptorul EGA, deoarece adaptorul VGA, având performanțe superioare, permite utilizarea acestor facilități.

Numărul maxim al culorilor care pot fi afișate cu ajutorul unui adaptor EGA este de 64.

Culorile se codifică prin numere întregi din intervalul  $[0,63]$ .

Cele 64 de culori nu pot fi afișate simultan pe ecran. În cazul adaptorului EGA se pot afișa simultan pe ecran cel mult 16 culori. Mulțimea culorilor care pot fi afișate simultan pe ecran se numește *paletă*. Culorile din componența unei palete pot fi modificate de utilizator prin intermediul funcțiilor standard. La inițializarea modului grafic se setează o paletă *implicită*.

Paleta se definește cu ajutorul unui tablou de 16 elemente pentru adaptorul EGA. Elementele acestui tablou au valori din intervalul  $[0,63]$ . Fiecare element din acest tablou reprezintă codul unei culori.

Codurile culorilor din paleta implicită au denumiri simbolice definite în fișierul *graphics.h*. Ele au prefixul EGA\_

În tabela de mai jos se indică codurile culorilor pentru paleta implicită.

Funcțiile de gestiune a culorilor pot avea ca parametri nu numai codurile culorilor, ci și indecși în tabloul care definește culorile unei palete. De aceea, indecși din intervalul  $[0,15]$  pot fi referiți prin constante simbolice definite în fișierul *graphics.h*. Aceste denumiri sugerează culoarea din compunerea paletei.

**Tabela paletelor implicite**

indice		codul culorilor	
denumire simbolică	valoare	denumire simbolică	valoare
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_BROWN	20
LIGHTGRAY	7	EGA_LIGHTGRAY	7
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Culoarea fondului (*background*) este totdeauna cea corespunzătoare indicelui zero.

Culoarea pentru desenare (*foreground*) este cea corespunzătoare indicelui 15.

Culoarea de fond poate fi modificată cu ajutorul funcției *setbkcolor*. Aceasta are prototipul:

```
void far setbkcolor(int culoare);
```

unde:

*culoare* - Este index în tabloul care definește paleta.

De exemplu, dacă se utilizează apelul:

```
setbkcolor(BLUE);
```

atunci culoarea de fond devine albastră

Pentru a cunoaște culoarea de fond curentă se poate apela funcția *getbkcolor* de prototip:

**int far getbkcolor(void);**

Ea returnează indexul în tabloul care definește paleta pentru culoarea de fond.

Culoarea pentru desenare poate fi modificată folosind funcția *setcolor* de prototip:

**void far setcolor(int culoare);**

unde:

*culoare* - Este index în tabloul care definește paleta.

De exemplu, dacă se utilizează apelul:

**setcolor(YELLOW);**

atunci culoarea pentru desenare este galbenă.

Culoarea pentru desenare se poate determina apelind funcția *getcolor* de prototip:

**int far getcolor(void);**

Ea returnează indexul în tabloul care definește paleta relativ la culoarea pentru desenare.

Paleta curentă poate fi modificată folosind funcțiile *setpalette* și *setallpalette*.

Prima se folosește pentru a modifica o culoare din paleta curentă. Ea are prototipul:

**void far setpalette(int index, int cod);**

unde:

*index* - Este un întreg din intervalul [0,15] și reprezintă indexul în tabloul care definește paleta pentru culoarea care se modifică.

*cod* - Este un întreg din intervalul [0,63] și reprezintă codul culorii care o înlocuiește în paletă pe cea veche.

De exemplu, apelul:

**setpalette(DARKGRAY,45);**

modifică culoarea corespunzătoare indicelui DARKGRAY (adică 8), prin culoarea de cod 45.

Cealaltă funcție permite modificarea simultană a mai multor culori din compunerea paletelor. Ea are prototipul:

```
void far setallpalette(struct palettetype far *paleta);
```

unde:

*palettetype* - Este un tip definit în fișierul *graphics.h* ca mai jos:

```
struct palettetype {
    unsigned char size;
    signed char colors[MAXCOLORS + 1];
};
```

unde:

*size* - Este dimensiunea paletei.

*colors* - Este un tablou ale cărui elemente au ca valori codurile culorilor componente ale paletei care se definește.

Modificarea paletei curente cu ajutorul funcției *setpalette* sau *setallpalette* conduce la schimbarea corespunzătoare a culorilor afișate pe ecran în momentul apelului funcțiilor respective.

Pentru a determina codurile culorilor componente ale paletei curente vom folosi funcția *getpalette* de prototip

```
void far getpalette(struct palettetype far *paleta);
```

**Exemplu:**

```
struct palettetype pal;
...
getpalette(&pal);
...
```

După apelul funcției *getpalette* se atribuie componentelor structurii datele corespunzătoare din paleta curentă.

Paleta implicită poate fi determinată folosind funcția *getdefaultpalette* de prototip:

```
struct palettetype *far getdefaultpalette(void);
```

Numărul culorilor dintr-o paletă poate fi obținut apelînd funcția *getmaxcolor* de prototip:

```
int far getmaxcolor(void);
```

Funcția returnează numărul maxim de culori diminuat cu 1. Deci, în cazul adaptorului EGA funcția returnează valoarea 15.

O altă funcție care determină dimensiunea paletei este funcția *getpalettesize*. Ea are prototipul:

```
int far getpalettesize(void);
```

Funcția returnează numărul culorilor componente ale paletei. În cazul adaptorului EGA funcția returnează valoarea 16.

**Exerciții:**

19.2 Să se scrie un program care afișează codurile culorilor pentru paleta implicită.

## PROGRAMUL BXIX2

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

main() /* afiseaza codurile culorilor pentru paleta implicita */
{
    int gd=DETECT, gm, i;
    struct palettetype far *pal=(void *)0;

    initgraph(&gd, &gm, "c:\\borlandc\\bgi");
    pal=getdefaultpalette();
    for(i=0; i<16; i++){
        printf("colors[%d]=%d\n", i, pal->colors[i]);
        getch();
    }
    closegraph();
}
```

## 19.3. Starea ecranului

În mod grafic, ecranul se compune din  $n \times m$  puncte luminoase (pixeli). Aceasta înseamnă că pe ecran se pot afișa  $m$  linii a  $n$  pixeli fiecare.

Poziția unui pixel se definește printr-un sistem binar de întregi

( $x, y$ )

numite coordonatele pixelului. Coordonata  $x$  definește coloana pixelului, iar  $y$  definește linia acestuia.

Pixelul aflat în colțul din stînga sus are coordonatele (0,0).

Coloanele se numără de la stînga spre dreapta, iar liniile de sus în jos.

Biblioteca grafică a sistemului conține 4 funcții care permit utilizatorului să obțină următoarele informații relativ la ecran:

- coordonata maximă pe orizontală,
- coordonata maximă pe verticală;
- poziția curentă (pixel curent).

Prototipurile acestor funcții sînt:

`int far getmaxx(void);`

funcția returnează coordonata maximă pe orizontală (abscisa maximă);

`int far getmaxy(void);`

funcția returnează coordonata maximă pe verticală (ordonata maximă);

`int far getx(void);`

funcția returnează poziția pe orizontală (abscisa) a pixelului curent;

`int far gety(void);`

funcția returnează poziția pe verticală (ordonata) a pixelului curent.

**Exerciții:**

19.3 Să se scrie un program care afișează următoarele informații:

- culoarea fondului;
- culoarea pentru desenare;
- coordonatele maxime;
- coordonatele pixelului curent.

## PROGRAMUL BXIX3

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
```

```
main() /* afiseaza:
```

- culoarea fondului;
- culoarea pentru desenare;
- coordonatele maxime;
- coordonatele pixelului curent. \*/

```
{
    int gd=DETECT, gm, culf, culd, crtx, crty, maxx, maxy;
```

```

initgraph(&gd,&gm,"c:\\borlandc\\bgi");
culf=getbkcolor();
culd=getcolor();
maxx=getmaxx();
maxy=getmaxy();
crtx=getx();
crtx=gety();
closegraph();
printf("culoarea fondului=%d\n",culf);
printf("culoarea pentru desenare=%d\n",culd);
printf("abscisa maxima=%d\n",maxx);
printf("ordonata maxima=%d\n",maxy);
printf("abscisa curenta=%d\n",crtx);
printf("ordonata curenta=%d\n",crtx);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();
}

```

## 19.4. Gestiunea textelor

Afișarea textelor presupune definirea unor parametri care pot fi gestionați prin funcțiile descrise mai jos.

În mod grafic dispunem de mai multe seturi de caractere. Setul de caractere se alege prin intermediul parametrului numit *font*. Pentru acest parametru se pot utiliza următoarele valori.

Constantă simbolică	Valoare
DEFAULT_FONT`	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SERIF_FONT	3
GOTHIC_FONT	4

Alți parametri utilizați în definirea caracterelor sint:

- înălțimea și lățimea caracterelor;
- direcția de scriere a caracterelor;
- de la stînga la dreapta: HORIZ\_DIR;

- de jos în sus: **VERT\_DIR**;
- cadrajul caracterelor față de poziția curentă:
- pe orizontală; poziția curentă se află.
  - » în stînga: **LEFT\_TEXT**;
  - » în centru: **CENTER\_TEXT**;
  - » în dreapta: **RIGHT\_TEXT**.
- pe verticală, poziția curentă se afla în:
  - » marginea inferioară: **BOTTOM\_TEXT**;
  - » centru: **CENTER\_TEXT**;
  - » marginea superioară. **TOP\_TEXT**.

Acești parametri se setează cu ajutorul a două funcții *settextstyle* și *settextjustify*. Prima are prototipul.

**void far settextstyle(int font, int direction, int charsize);**

unde:

- |                  |   |
|------------------|---|
| <i>font</i>      | - Definește setul de caractere.                 |
| <i>direction</i> | - Definește direcția de scriere a textului.     |
| <i>charsize</i>  | - Definește dimensiunea caracterului în pixeli. |

Dimensiunea se definește astfel:

Valoarea parametrului	Matricea folosită pentru afișarea caracterului (în pixeli)
1	8*8
2	16*16
3	24*24
...	...
10	80*80

Dimensiunea poate fi stabilită de utilizator folosind funcția *setuserchar-size*. În acest caz, parametrul *charsize* are valoarea zero.

Cea de-a doua funcție definește cadrajul textului. Ea are prototipul

**void far settextjustify(int oriz, int vert),**

unde:

- |             |                                     |
|-------------|-------------------------------------|
| <i>oriz</i> | - Definește cadrajul pe orizontală; |
| <i>vert</i> | - Definește cadrajul pe verticală.  |



Valorile acestor parametri pot fi determinate cu ajutorul funcției *gettextsettings* de prototip:

```
void far gettextsettings(struct textsettingstype far *textinfo);
```

Tipul *textsettingstype* este definit în fișierul *graphics.h* astfel:

```
struct textsettingstype {  
    int font;  
    int direction;  
    int charsize;  
    int horiz;  
    int vert;  
};
```

Funcția *gettextsettings* atribuie componentelor structurii de tip *textsettingstype* valorile curente.

Amintim valorile numerice ale constantelor simbolice indicate mai sus.

Constantă simbolică	Valoare
LEFT_TEXT	0
CENTER_TEXT	1
RIGHT_TEXT	2
BOTTOM_TEXT	0
TOP_TEXT	2
HORIZ_DIR	0
VERT_DIR	1

După setarea parametrilor de mai sus se pot afișa texte folosind funcțiile *outtext* și *outtextxy*. Prima afișează textul începînd cu poziția curentă de pe ecran. Cea de a doua funcție permite afișarea textului începînd cu un punct al cărui coordonate sînt definite prin primii doi parametri efectivi ai funcției.

Funcțiile afișează caractere colorate folosind culoarea de desenare curentă.

Funcția *outtext* are prototipul:

```
void far outtext(char far *sir);
```

unde:

*sir* Este pointer spre o zonă de memorie în care se păstrează caracterele de afișat.

Se afișează caracterele respective până la întâlnirea caracterului NUL.

Funcția *outtextxy* are prototipul:

```
void far outtextxy(int x, int y, char far *sir);
```

unde:

- (x,y) - Definește poziția punctului de pe ecran începând cu care se afișează textul.
- sir - Are aceeași semnificație ca în cazul funcției *outtext*.

Dimensiunile în pixeli a unui șir de caractere se pot determina folosind funcțiile *textheight* și *textwidth*. Acestea au prototipurile:

```
int far textheight(char far *sir);
```

funcția returnează înălțimea în pixeli a șirului păstrat în zona spre care pointează *sir*;

```
int far textwidth(char far *sir);
```

funcția returnează lățimea în pixeli a șirului aflat în zona spre care pointează *sir*.

Utilizatorul poate defini dimensiunea caracterelor apelând funcția *setusercharsize*. Ea are prototipul:

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

Dimensiunea caracterelor se definește prin înmulțirea lățimii lor cu *multx/divx* și a înălțimii cu *multy/divy*.

Modificarea dimensiunii caracterelor în acest mod este posibilă pentru fonturile diferite de fontul `DEFAULT_FONT`.

#### Observație:

Funcția *printf* poate fi folosită pentru a afișa caractere în mod obișnuit. Ea ignoră parametri indicați mai sus.

#### Exerciții:

19.4 Să se scrie un program care afișează texte folosind toate cele 5 fonturi, caracterele având pe rând dimensiunile 1, 2, 3 și 4. În cazul fonturilor diferite de `DEFAULT_FONT`, se vor afișa texte ale căror caractere se vor afla în rapoartele:

- 4/3 în lățime;
- 2/1 în înălțime.

## PROGRAMUL BXIX4

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>

main() /* afiseaza texte cu diferite fonturi si dimensiuni */
{
    int gdriver=DETECT,gmod;
    char *denfont[]={
        "Val=0 DEFAULT_FONT",
        "Val=1 TRIPLEX_FONT",
        "Val=2 SMALL_FONT",
        "Val=3 SANS_SERIF_FONT",
        "Val=4 GOTHIC_FONT"
    };
    int stil,x=0,y=0;
    int dim;
    char dimensiune[30];

    for(dim=1;dim<5;dim++){
        y=0;
        initgraph(&gdriver,&gmod,"c:\\borlandc\\bgi");
        for(stil=DEFAULT_FONT;stil<=GOTHIC_FONT;stil++){
            {
                settextstyle(stil,HORIZ_DIR,dim);
                sprintf(dimensiune,"dim=%d font:",dim);
                outtextxy(x,y,dimensiune);
                x += textwidth(dimensiune); /* avans la coloana
                                                libera de pe aceeași linie */
                outtextxy(x,y,denfont[stil]);
                y += textheight(denfont[stil]); /* avans la
                                                inceputul liniei următoare */
                x=0;
            }
            /* se definește dimensiunea de către utilizator
            raport: 4/3 în lățime;
                2/1 în lungime. */
            if(stil!=DEFAULT_FONT){
                setusercharsize(4,3,2,1); /* definește
                raporturile pentru dimensiunile caracterelor */
                strcpy(dimensiune,
                    "dim utilizator:4/3,2/1");
            }
        }
    }
}
```

```

        outtextxy(x,y,dimensiune);
        y+=textheight(dimensiune);
        x=0;
    }
    getch();
}
closegraph();
printf("Actionati o tasta pentru a\
continua\n");
getch();
}
}

```

19.5 Să se scrie un program care afișează texte cadrate in toate variantele definite de funcția *settextjustify*.

### PROGRAMUL BXIX5

```

#include <graphics.h>
#include <conio.h>
#include <stdio.h>

main() /* afiseaza texte cadrate in toate variantele definite de functia
        settextjustify */
{
    int gdriver=DETECT,gmod;
    char *hjust[]={ "LEFT_TEXT",
        "CENTER_TEXT",
        "RIGHT_TEXT",
    };
    char *vjust[]={ "BOTTOM_TEXT",
        "CENTER_TEXT",
        "TOP_TEXT"
    };
    int x=200,y=100,hj,vj;

    for(hj=LEFT_TEXT;hj<=RIGHT_TEXT;hj++) {
        initgraph(&gdriver,&gmod,"c:\\borlandc\\bgi");
        outtextxy(x,y,hjust[hj]);
        y+=textheight(hjust[hj])+8;
        for(vj=BOTTOM_TEXT;vj<=TOP_TEXT;vj++){
            settextjustify(hj,vj);
            outtextxy(x,y,"N");

```

```

        x+=textwidth("N");
        outtextxy(x+100*vj+100,y,vjust[vj]);
        getch();
    }
    closegraph();
}
}

```

## 19.5. Gestiunea imaginilor

În paragrafele precedente s-a arătat că ecranul în mod grafic se compune din  $n*m$  puncte luminoase numite *pixel*. Un pixel are o poziție definită prin coordonatele sale și este colorat în cazul adaptoarelor color.

La adaptoarele monocrom pixelul are o nuanță de gri.

Utilizatorul poate afișa pe ecran un pixel cu ajutorul funcției *putpixel*. Aceasta are prototipul:

```
void far putpixel(int x, int y, int culoare);
```

unde:

- |                |   |
|----------------|---|
| <i>(x,y)</i>   | - Definește poziția punctului.  |
| <i>culoare</i> | - Definește culoarea punctului.<br>- Este un întreg din intervalul [0,15] și reprezintă indicele culorii în tabela care definește paleta curentă. |

Funcția *getpixel* permite stabilirea culorii unui pixel afișat pe ecran. Ea are prototipul:

```
unsigned far getpixel(int x, int y);
```

unde:

- |              |                                |
|--------------|--------------------------------|
| <i>(x,y)</i> | - Definește poziția punctului. |
|--------------|--------------------------------|

Funcția returnează un întreg din intervalul [0,15]. Acesta definește culoarea pixelului de coordonate *(x,y)*, fiind index în tabloul care definește paleta curentă.

Ecranul poate fi partajat în mai multe părți care pot fi gestionate independent. Aceste părți le vom numi *ferestre grafice*.

În continuare, prin fereastră vom înțelege o fereastră grafică. O fereastră se definește cu ajutorul funcției *setviewport* de prototip:

```
void far setviewport(int st, int sus, int dr, int jos, int d),
```

unde:

- (st,sus)* - Sînt coordonatele colțului stînga sus al ferestrei
- (dr,jos)* - Sînt coordonatele colțului dreapta jos al ferestrei.
- d* - Indicator cu privire la decuparea desenului (vezi mai jos).

Fereastra definită în urma apelului funcției *setviewport* devine fereastra activă. Inițial (imediat după setarea modului grafic), fereastra activă este tot ecranul.

O fereastră activă se poate șterge cu ajutorul funcției *clearviewport*. Ea are prototipul:

```
void far clearviewport(void);
```

După apelul funcției *clearviewport*, toți pixelii ferestrei active au aceeași culoare și anume culoarea de fond curentă.

Poziția curentă după apelul funcției *clearviewport* este pixelul de coordonate relative (0,0), adică chiar colțul din stînga sus al ferestrei.

O altă funcție utilizată pentru a șterge tot ecranul este funcția *cleardevice*. Ea are prototipul:

```
void far cleardevice(void);
```

După apelul funcției *cleardevice* se șterge tot ecranul și pixelul curent devine cel din colțul stînga sus al ecranului.

Parametri ferestrei active se pot determina apelînd funcția *getviewsettings*. Aceasta are prototipul:

```
void far getviewsettings(struct viewporttype far *fereastra),
```

unde:

*viewporttype* - Este un tip definit în fișierul *graphics.h* astfel:

```
struct viewporttype {  
    int left;  
    int top;  
    int right;  
    int bottom;  
    int clip;  
};
```

După apelul funcției *getviewsettings*, la componentele structurii de tip *viewporttype* de la apel li se atribuie valorile corespunzătoare ale ferestrei active.

Parametrul *clip* are două valori:

CLIP\_ON (valoarea 1)

și

CLIP\_OFF (valoarea zero).

Dacă *clip* are valoarea 1, atunci funcțiile de afișare a textelor și de desenare nu pot scrie sau desena în afara limitelor ferestrei active. În caz contrar, se pot depăși limitele ferestrei active. Deci textele și figurile care nu încap în fereastra activă se trunchiază dacă:

*clip* = CLIP\_ON.

Imaginea ecranului se păstrează în memoria video a adaptorului grafic și formează o pagină. În cazul adaptoarelor de tip EGA/VGA, adaptorul dispune de o memorie video capabilă să memoreze 8 pagini. Acestea se numerotează de la 0 la 7.

Funcțiile de desenare și scriere de texte acționează asupra unei singure pagini. Aceasta se numește pagina activă. Utilizatorul poate activa o pagină folosind funcția *setactivepage* de prototip:

```
void far setactivepage(int nrpag);
```

unde:

*nrpag* - Este numărul paginii care se activează

De obicei, pagina activă este vizualizată pe ecran. Cu toate acestea, programatorul are posibilitatea să vizualizeze o altă pagină decît cea activă. Aceasta se realizează utilizînd funcția *setvisualpage* de prototip:

```
void far setvisualpage(int nrpag);
```

unde:

*nrpag* - Este numărul paginii care se vizualizează.

Această funcție poate fi utilă pentru animație.

Imaginea unei zone dreptunghiulare de pe ecran poate fi salvată în memorie folosind funcția *getimage*. Ea are prototipul:

```
void far getimage(int st, int sus, int dr, int jos, void far *zf);
```

unde:

(*st,sus*) - Definește coordonatele colțului stînga sus a zonei de pe ecran care se salvează.

(*dr,jos*) - Definește coordonatele colțului dreapta jos a zonei de pe ecran care se salvează.

*zt* - Pointer spre zona de memorie în care se salvează imaginea de pe ecran.

Dimensiunea zonei de memorie spre care pointează *zt* trebuie să fie suficient de mare pentru a putea salva datele care definesc imaginea de pe ecran, care se salvează. Această dimensiune se poate determina folosind funcția *imagesize* de prototip:

```
unsigned far imagesize(int st, int sus, int dr, int jos);
```

unde:

- (*st,sus*) - Definește coordonatele colțului din stînga sus a zonei dreptunghiulare de pe ecran.
- (*dr,jos*) - Definește coordonatele colțului din dreapta jos a zonei dreptunghiulare de pe ecran.

Funcția *imagesize* se apelează înainte de a apela funcția *getimage* pentru a stabili dimensiunea zonei de memorie necesară pentru a salva o imagine dreptunghiulară de pe ecran. Zona de memorie respectivă se poate rezerva în memoria heap ținînd seama de valoarea returnată de funcția *imagesize*.

Imaginea de pe ecran salvată cu ajutorul funcției *getimage*, poate fi afișată pe ecran în orice parte a acestuia, cu ajutorul funcției *putimage*. Cu această ocazie se pot face anumite operații asupra datelor care definesc imaginea. Funcția are prototipul:

```
void far putimage(int st, int sus, void far *zt, int op);
```

unde:

- (*st,sus*) - Definește coordonatele colțului stînga sus a zonei de pe ecran în care se afișează imaginea.
- zt* - Pointer spre zona în care se păstrează datele care formează imaginea de afișat.
- Aceste date au fost păstrate în zona respectivă prin intermediul funcției *getimage*.
- op* - Definește operația între datele aflate în zona spre care pointează *zt* și cele existente pe ecran în zona dreptunghiulară definită de parametri *st, sus* (colțul din stînga sus; colțul din dreapta jos rezultă din datele existente în zona spre care pointează *zt*).

Parametrul *op* se definește ca mai jos:



Constantă simbolică	Valoare	Acțiune
COPY_PUT	0	copiază imaginea din memorie pe ecran
XOR_PUT	1	"sau exclusiv" între datele de pe ecran și cele aflate în memorie
OR_PUT	2	"sau" între datele de pe ecran și cele din memorie
AND_PUT	3	"și" între datele de pe ecran și cele din memorie
NOT_PUT	4	copiază imaginea din memorie pe ecran complementind datele aflate în memorie.

Pentru a defini pixelul curent se utilizează funcția *moveto*. Aceasta are prototipul:

```
void far moveto(int x, int y);
```

După apelul funcției *moveto*, pixelul curent devine cel de coordonate (x,y).

O funcție înrudită cu aceasta este funcția *moverel*. Aceasta are prototipul:

```
void far moverel(int dx, int dy);
```

Dacă notăm cu (x,y) coordonatele pixelului curent, atunci după apelul funcției *moverel*, pixelul curent are coordonatele:

```
(x+dx, y+dy)
```

**Observație:**

Majoritatea funcțiilor care au ca parametri coordonate de pixel, interpretează aceste coordonate ca fiind relative față de fereastra activă al cărei colț stnga sus are coordonatele(0,0). Așa de exemplu, funcțiile:

```
outtextxy, putpixel, getpixel, moveto
```

au ca parametri coordonate relative la pixelul din colțul stnga sus al ferestrei active.

Funcțiile *getx* și *gety* returnează abscisa, respectiv ordonata, relative la pixelul din colțul stnga sus al ferestrei active.

În schimb funcțiile *getmaxx* și *getmaxy* returnează totdeauna abscisa, respectiv ordonata, maximă pentru tot ecranul în conformitate cu modul

grafic.

De asemenea, funcțiile *setviewport* și *getviewsettings* gestionează coordonate care sînt relative față de colțul stînga sus al ecranului și nu față de fereastra activă. Astfel de coordonate le vom numi în continuare *absolute*.

### Exerciții:

19.6 Să se scrie un program care realizează următoarele:

- a. Afișează, într-o zonă de dimensiune 50\*50, pixeli colorați folosind toate culorile din paletă.

Zona se află în colțul din stînga sus al ecranului.

- b. Definiște fereastra de coordonate:

(60,0) - colțul stînga sus;

(120,60) - colțul dreapta jos.

Afișează în fereastra activă pixeli colorați folosind toate culorile din paletă.

- c. Definiște fereastra de coordonate:

(20,100) - colțul stînga sus;

(100,180) - colțul dreapta jos.

Afișează în fereastra activă pixeli colorați folosind toate culorile din paletă.

Afișează în fereastra activă textul "abcdefg" începînd cu poziția de coordonate(1,1).

Se utilizează culoarea de index 6.

- d. Se șterge fereastra activă.

Se afișează textul "abcdefg" începînd cu poziția curentă din fereastră și folosind culoarea de index 14.

- e. Se șterge tot ecranul.

- f. Se revine din modul grafic.

După realizarea fiecărui punct de mai sus se vizualizează ecranul apelînd funcția *getch*. Pentru a continua, se acționează o tastă oarecare.

### PROGRAMUL BXIX6

```
#include <graphics.h>
```

```

#include <conio.h>

main() /* - afiseaza in trei zone ale ecranului pixeli colorati folosind
        toate culorile paletii;
        - in una din zone se afiseaza si textul "abcdefg" ;
        - in final se sterg zonele afisate. */
{
    int gd=DETECT,gm;
    int i,j,c;

    initgraph(&gd,&gm,"c:\\borlandc\\bgi");

    /* zona din coltul stinga sus al ecranului */
    for(i=0;i<50;i++){
        c=i;
        for(j=0;j<50;j++,c++){
            c=c%16;
            putpixel(i,j,c);
        }
    }
    getch();

    /* defineste o fereastra si afiseaza in ea pixeli colorati */
    setviewport(60,0,120,60,1);
    for(i=0;i<50;i++){
        c=0;
        for(j=0;j<50;j++,c++){
            c=c%16;
            putpixel(i,j,c);
        }
    }
    getch();

    /* defineste ultima zona si afiseaza in ea pixeli colorati */
    setviewport(20,100,100,180,1);
    for(i=0;i<40;i++){
        c=10;
        for(j=0;j<250;j++,c++){
            c=c%16;
            putpixel(i,j,c);
        }
    }
    /* afiseaza textul "abcdefg" */

```

```

setcolor(6);
outtextxy(1,1,"abcdefg");
getch();

/* sterge ultima zona si apoi afiseaza acelasi text folosind culoarea de
index 14 */
clearviewport();
setcolor(14);
outtext("abcdefg");
getch();

/* sterge tot ecranul */
cleardevice();
getch();

/* iesire din modul grafic */
closegraph();
getch();
}

```

19.7 Să se scrie un program care realizează următoarele:

- Afișează un pixel în punctul de coordonate absolute (20,30), apoi textul.

(x,y)

unde:

x și y - Sînt coordonatele returnate de funcțiile *getx* și respectiv *gety*.

- Se deplasează poziția curentă cu valoarea relativă 30, atît pe abscisă, cît și pe ordonată, apoi se afișează pixelul curent și textul:

(x,y)

unde:

x și y - Sînt valorile returnate de funcțiile *getx* și respectiv *gety*.

- Se definește fereastra de coordonate:

(200,0,420,70);

apoi se repetă secvențele a-b de mai sus.

- Se șterge fereastra activă.

- Se șterge ecranul și se iese din modul grafic.

Culoarea pentru afişare este culoarea de index maxim.

## PROGRAMUL BXIX7

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>

main() /* -afiseaza:
    a- un pixel in punctul de coordonate absolute (20,30),
        apoi textul
            (x,y)
        unde x si y sint coordonatele returnate de getx si res-
        pectiv gety;
    b- se deplaseaza pozitia curenta cu valoarea relativa
        30 atit pe abscisa cit si pe ordonata, apoi se afiseaza
        pixelul curent si textul
            (x,y)
        unde x si y se definesc ca mai sus;
    c- se defineste fereastra
        200,0,420,70
        si se repeta secventele a-b de mai sus;
    se sterg afisarile si apoi se iese din modul grafic. */
{
    int gd=DETECT,gm;
    int x,y,i;
    char msg[80];

    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    for(i=0;i<2;i++){
        moveto(20,30);
        putpixel(getx(),gety(),getmaxcolor());
        sprintf(msg,"%d %d",getx(),gety());
        outtext(msg);

/* deplasare relativa fata de pozitia curenta */
        moverel(30,30);
        putpixel(getx(),gety(),getmaxcolor());
        sprintf(msg,"%d %d",getx(),gety());
        outtext(msg);
        getch();
        setviewport(200,0,420,70,1);
    }
}
```

```

clearviewport();
getch();
cleardevice();
getch();
closegraph();
}

```

19.8 Să se scrie un program care realizează următoarele:

- Afișează pixeli colorați într-o zonă dreptunghiulară în colțul din stînga sus al ecranului, apoi o afișează pe ecran în zone ce au poziții definite aleator.
- La afișarea imaginii se folosesc toate operațiile dintre imagini oferite de funcția *putimage*.
- Culoarea de fond se schimbă folosind toate culorile din paletă.

După afișările indicate mai sus execuția se poate termina acționînd tasta zero; orice altă tastă acționată permite continuarea programului cu un nou set de imagini afișate aleator.

Se observă efectul operațiilor dintre imagini cînd acestea se suprapun.

## PROGRAMUL BXIX8

```

#include <graphics.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <dos.h>
#include <stdio.h>

```

```

main() /* - afiseaza pixeli colorati intr-o zona dreptunghiulara in
        coltul din stinga sus al ecranului, apoi afiseaza imaginea
        respectiva pe ecran in zone ce au pozitii definite aleator;
        - la afisarea imaginii se folosesc toate operatiile oferite de
        functia putimage;
        - de asemenea, se schimba culoarea de fond folosind toate
        culorile din paleta. */

```

```

{
    int gd=DETECT,gm;
    int i,j,c,k;
    unsigned dim;
    void far *buf;
    struct time t;

```

```

int rx,ry,x,y;

initgraph(&gd,&gm,"c:\\borlandc\\bgi");
/* afiseaza pixeli colorati */
for(i=0;i<50;i++){
    c=i;
    for(j=0;j<50;j++,c++) {
        c=c%16;
        putpixel(i,j,c);
    }
}
getch();

/* salveaza zona afisata pe ecran */
dim=imagesize(0,0,50,50);
if((buf=farmalloc(dim))==0){
    closegraph();
    printf("Memorie insuficienta\n");
    exit(1);
}
getimage(0,0,50,50,buf);

/* seteaza saminta */
gettime(&t);
srand(t.ti_hour*3600L +t.ti_min*60+t.ti_sec);

/* se determina valorile maxime pentru abscisa si ordonata coltului sting
al imaginilor */
x=getmaxx()-50;
y=getmaxy()-50;

/* afisarea aleatoare a imaginilor */
do{
    for(k=0;k<16;k++) {
        setbkcolor(k);
        for(i=COPY_PUT;i<=NOT_PUT;i++){
            rx=random(x);
            ry=random(y);
            setviewport(0,0,getmaxx(),getmaxy(),1);
            putimage(rx,ry,buf,i);
            getch();
        }
    }
}

```

```

    setcolor(1); /* culoarea pentru afisarea textului in partea
                  de jos a ecranului */
    setviewport(1,340,639,349,1); /* fereastra pentru
                                     text */
    outtextxy(0,0,"Pentru a termina tastati\
              zero; se continua cu orice alta\
              tasta");
    if(getch()=='0')
        break;
    clearviewport(); /* sterge textul afisat */
}while(1);
closegraph();
}

```

## 19.6. Tratarea erorilor

Erorile survenite in gestiunea in mod grafic a ecranului pot fi puse in evidenta cu ajutorul functiei *graphresult*. Ea are prototipul:

```
int far graphresult(void);
```

Functia returneaza codul ultimei erori care a apărut inaintea apelului functiei *graphresult* sau valoarea constantei simbolice *grOk* daca nu au fost erori.

Constanta *grOk* este definita in fisierul *graphics.h*.

Mesajul de eroare poate fi decodificat cu ajutorul functiei *grapherrormsg*. Aceasta are prototipul:

```
char far *far grapherrormsg(int coderoare),
```

unde:

*coderoare* - Este valoarea returnata de functia *graphresult*.

Functia *grapherrormsg* returneaza un pointer spre textul de eroare.

Constanta *grOk* are valoarea zero. Codurile de eroare au valori negative.

### Exerciții:

199 Să se scrie un program care afișează parametri implicați ai modului grafic setat prin apelul functiei *initgraph*:

- numele adaptorului grafic;
- numele modului grafic;
- rezoluția;



- fereastra curentă;
- decupajul;
- poziția curentă;
- numărul culorilor disponibile;
- culoarea curentă;
- setul de caractere;
- direcția textului;
- dimensiunea caracterelor;
- cadrajul textelor.

## PROGRAMUL BXIX9

```
#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>

int grafdriver, grafmod;
struct palettetype paleta;
int maxculori;
int xmax, ymax;
int posx, posy;
char *Fonts[]={"DefaultFont", "TriplexFont",
               "SmallFont", "SansSerifFont",
               "GothicFont"};
char *TextDirect[]={ "HorizDir", "VertDir" };
char *HorizJust[]={ "LeftText", "CenterText",
                   "RightText" };
char *VertJust[]={ "BottomText", "CenterText",
                  "TopText" };

void inimodgrafic() /* initializeaza modul grafic */
{
    int eroare;

    grafdriver=DETECT;
    initgraph( &grafdriver, &grafmod,
               "c:\\borlandc\\bgi");
    eroare=graphresult();
    if(eroare!= grOk){
        printf ("eroare la initializarea modului\
```

```

        grafic: ts\n",
        grapherrormsg(eroare));
        exit(1);
    }
    getpalette(&paleta);
    maxculori=getmaxcolor()+1;
    xmax=getmaxx();
    ymax=getmaxy();
    posx=getx();
    posy=gety();
}

void fereastraprincipala(char *antet)
/* afiseaza antetul in fereastra principala */
{
    int inalt;

    /* sterge ecranul */
    cleardevice();

    /* seteaza fereastra principala pe tot ecranul */
    setviewport (0,0,xmax,ymax,1);

    /* determina inaltimea textului */
    inalt=textheight("H");

    /* afisarea textului */
    outtextxy(xmax/2, 2, antet );

    /* seteaza o fereastra pentru a continua afisarea */
    setviewport(1, inalt+5, xmax-1,
                ymax-(inalt+ 5),1);
}

void afisinit()
/* afiseaza datele de la initializarea modului grafic */
{
    struct viewporttype infview;
    struct textsettingstype inftext;
    char *driver,*mod;
    int x,y;
    char sir[130];

```

```

getviewsettings(&infview);

gettextsettings(&inftext);
driver=getdrivername();
mod=getmodename(grafmod);

x=10;
y=4;

/* se scrie textul:rezultatele initializarii */
fereastraprincipala("Rezultatele initializarii");

/* cadrajul textului */
settextjustify(LEFT_TEXT,TOP_TEXT);

/* afiseaza adaptorul grafic */
sprintf(sir,"adaptorul grafic: %-20s(%d)",
        driver, grafdriver);
outtextxy(x,y,sir);
y+=8;

/* afiseaza modul grafic */
sprintf(sir, "modul grafic: %-20s (%d)", mod,
        grafmod);
outtextxy(x,y,sir);
y+=8;

/* afiseaza rezolutia */
sprintf(sir,"rezolutia: (0,0,%d,%d)", xmax, ymax);
outtextxy(x,y,sir);
y+=8;

/* fereastra grafica curenta */
sprintf(sir,"fereastra curenta:(%d, %d, %d, %d)",
        infview.left, infview.top,
        infview.right, infview.bottom);
outtextxy(x,y,sir);
y+=8;

sprintf(sir,"decupaj: %s",
        infview.clip ? "ON" : "OFF");
outtextxy(x,y, sir );
y+=8;

```

```

/* pozitia curenta */
sprintf(sir, "pozitia curenta: ( %d, %d )", posx,
        posy);
outtextxy( x, y, sir );
y+=8;

/* numar culori disponibile */
sprintf(sir, "numar culori: %d", maxculori);
outtextxy(x, y, sir );
y+=8;

/* culoarea curenta */
sprintf(sir, "culoarea curenta: %d", getcolor());
outtextxy(x, y, sir );
y+=8;

/* setul de caractere */
sprintf(sir, "setul de caractere: %s",
        Fonts[inftext.font]);
outtextxy(x, y, sir);
y+=8;

/* directia textului */
sprintf (sir, "directia textului: %s",
        TextDirect[inftext.direction]);
outtextxy(x, y, sir);
y+=8;

/* dimensiunea caracterului */
sprintf (sir, "dimensiunea caracterului: %d",
        inftext.charsize);
outtextxy(x, y, sir);
y+=8;

/* cadraj orizontal */
sprintf(sir, "cadraj orizontal: %s",
        HorizJust[inftext.horiz]);
outtextxy(x, y, sir);
y+=8;

/* cadraj vertical */
sprintf(sir, "cadraj vertical: %s",
        VertJust[inftext.vert]);

```

```

    outtextxy(x,y,sir);
}

main() /* afiseaza starea curenta a unor parametri ai modului grafic */
{
    inimodgrafic ();
    afisinit();
    outtextxy(8,ymax-50,"Actionati o tasta pentru a\
        termina");
    getch ();
    closegraph();
}

```

## 19.7. Desenare și colorare

Biblioteca standard a sistemului pune la dispoziția utilizatorului o serie de funcții care permit desenarea și colorarea unor figuri geometrice.

Amintim că un punct colorat (pixel) se afișează cu ajutorul funcției *putpixel* de prototip (vezi paragraful 19.5)

```
void far putpixel(int x, int y, int culoare);
```

unde.

(x,y) - Sînt coordonatele punctului care se afișează și ele sînt relative la fereastra activă.

culoare - Este index în tabloul care definește paleta curentă.

Indexul respectiv definește codul culorii pentru afișarea pixelului pe ecran.

Menționăm că în acest paragraf prin parametrul *culoare* vom înțelege un index în tabloul care definește paleta curentă.

Acest index definește codul culorii pentru desenarea și colorarea figurilor.

Pentru trasarea liniilor se pot folosi trei funcții: *line*, *lneto* și *linere1*.

Funcția *line* are prototipul:

```
void far line(int xstart, int ystart, int xfin, int yfin);
```

Funcția trasează un segment de dreapta ale cărui capete sînt punctele de coordonate.

(xstart,ystart)

și

(xfin,yfin).

Funcția *lineto* are prototipul:

**void far lineto(int x, int y);**

Ea trasează un segment de dreaptă care are ca origine poziția curentă, iar ca și punct final cel de coordonate (x,y).

Punctul final devine poziția curentă.

Amintim că funcția *moveto* permite definirea poziției curente (vezi paragraful 19.5).

Cea de a treia funcție utilizată la trasarea dreptelor are prototipul:

**void far linerel(int x, int y);**

Dacă notăm cu *xcr* și *ycr* coordonatele poziției curente, atunci funcția *linerel* trasează un segment de dreaptă ale cărui capete sînt punctele de coordonate:

(xcr,ycr)

și

(xcr+x,ycr+y).

Alte funcții care permit trasări de figuri geometrice utilizate frecvent sînt:

**void far arc(int xcentru, int ycentru, int unghistart, int unghifin, int raza);**

trasează un arc de cerc; unghiurile sînt exprimate în grade sexagesimale.

**void far circle(int xcentru, int ycentru, int raza);**

trasează un cerc; (xcentru,ycentru) sînt coordonatele centrului arcului de cerc și respectiv cercului trasat de aceste funcții, parametrul *raza* definește mărimea razei curbelor respective.

**void far ellipse(int xcentru, int ycentru,  
int unghistart, int unghifin,  
int semiaxamare, int semiaxamica);**

trasează un arc de elipsă cu centrul în punctul de coordonate (xcentru, ycentru) avînd semiaxa mare definită de parametrul *semiaxamare*, iar semiaxa mică de parametrul *semiaxamica*.

**void far rectangle(int st, int sus, int dr, int jos);**

trasează un dreptunghi definit de colțurile sale opuse:

- (*st,sus*) - Colțul din stînga sus.  
 (*dr,jos*) - Colțul din dreapta jos.

**void far\_drawpoly(int nr, int far \*tabpct);**

trasează o linie poligonală:

- nr* - Numărul laturilor.  
*tabpct* - Este un pointer spre întregi care definesc coordonatele vîrfurilor liniei poligonale.  
 - Acestea sînt păstrate sub forma: *abscisa\_i*, *ordonata\_i* unde *i* are valorile 1, 2, ..., *nr*+1.

Linia poligonală este închisă dacă primul punct coincide cu ultimul (au aceleași coordonate).

Coordonatele utilizate ca parametri la apelul acestor funcții sînt relative la fereastra activă.

Culoarea de trasare este cea curentă.

La trasarea liniilor cu ajutorul funcțiilor:

*line*, *lineto* și *lineret*

se poate defini un stil de trasare folosind funcția *setlinestyle*. Această funcție are prototipul:

**void far\_setlinestyle(int stil, unsigned sablon, int grosime);**

unde:

- stil* - Este întreg din intervalul [0,4] care definește stilul liniei conform tablei de mai jos:

Constantă simbolică	Valoare	Stil
SOLID_LINE	0	linie continuă
DOTTED_LINE	1	linie punctată
CENTER_LINE	2	linie întreruptă formată din liniuțe de două dimensiuni
DASHED_LINE	3	linie întreruptă formată din liniuțe de aceeași dimensiune
USERBIT_LINE	4	stil definit de utilizator prin șablon

- sablon* - Definește stilul liniei.  
 - Are sens numai cînd primul parametru (stil) are valoarea

4.

*grosime*

- În rest este neglijat și de aceea poate avea valoarea zero.
- Definește lățimea liniei în pixeli; pot fi două lățimi:

NORM\_WIDTH (valoarea 1 pixel)

și

THICK\_WIDTH (valoarea 3 pixeli).

Stilul curent poate fi determinat apelînd funcția *getlinesettings* de prototip:

```
void far getlinesettings(struct linesettingstype far *lineinfo);
```

unde:

*linesettingstype* - Este definit în fișierul *graphics.h* astfel:

```
struct linesettingstype {  
    int linestyle;  
    unsigned upattern;  
    int thickness;  
};
```

După apelul funcției *getlinesettings*, componentelor structurii de tip *linesettingstype* de la apel li se atribuie valorile curente ale stilului de trasare după cum urmează:

*linestyle* - Are ca valoare stilul definit mai sus.

*upattern* - Are ca valoare șablonul definit de utilizator. Această valoare are semnificație numai dacă *linestyle* are valoarea 4.

*thickness* - Are valoarea 1 sau 3 și reprezintă grosimea de trasare a dreptelor prin funcțiile *line*, *lineto* sau *lineref*.

Din cele de mai sus rezultă că pentru trasarea liniilor se pot folosi 4 stiluri *standard* (definite de valorile *SOLID\_LINE*, *DOTTED\_LINE*, *CENTER\_LINE* și *DASHED\_LINE*), precum și unul *nestandard*, definit de utilizator (valoarea *USERBIT\_LINE*).

Stilul *nestandard* se precizează cu ajutorul parametrului *sablon*. Acesta este o dată de tip *int* (16 biți). Fiecare bit din șablon reprezintă un pixel al liniei. Fiecare bit din șablon setat reprezintă un pixel colorat cu culoarea curentă, biții din șablon de valoarea zero reprezintă pixeli colorați cu culoarea de fond.

Alte funcții din biblioteca standard a sistemului permit trasarea de figuri



geometrice împreună cu colorarea interiorului acestora.

Indicăm mai jos prototipurile funcțiilor mai importante din această clasă:

```
void far_bar(int st, int sus, int dr, int jos);
```

unde:

*st, sus, dr și jos* - Au aceleași semnificații ca în cazul funcției *rectangle*.

Funcția desenează un domeniu dreptunghiular colorat (fără a avea o frontieră scoasă în evidență).

```
void far_bar3d(int st, int sus, int dr, int jos, int profunzime, int ind);
```

Funcția desenează o prismă dreptunghiulară colorată pentru *ind* diferit de zero. Pentru *ind=0*, nu se trasează partea de sus a prisme.

```
void far_pieslice(int xcentru, int ycentru, int unghistart,  
                 int unghifin, int raza);
```

Funcția desenează un sector de cerc colorat.

```
void far_fillpoly(int nr, int far *tabpct);
```

unde:

*nr și tabpct* - Au aceleași semnificații ca în cazul funcției *drawpoly*.

Funcția desenează un poligon colorat.

```
void far_fillellipse(int xcentru, int ycentru,  
                   int semiaxamare, int semiaxamica);
```

Funcția desenează o elipsă colorată.

Menționăm că figurile desenate cu funcțiile indicate mai sus se colorează folosind o culoare definită în prealabil cu ajutorul funcției *setfillstyle*. De asemenea, o figură poate fi colorată în mai multe feluri:

- folosind culoarea de fond;
- colorare uniformă (toți pixelii din interiorul figurii au aceeași culoare) folosind culoarea definită prin funcția *setfillstyle*;
- colorare prin hașură.

Hașura poate fi standard sau definită de utilizator.

Modul de colorare al unei figuri se definește tot cu ajutorul funcției *setfillstyle*. Ea are prototipul:

```
void far_setfillstyle(int hasura, int culoarea);
```

unde:

- hasura* - Definește modul de colorare conform tabelului de mai jos.
- culoarea* - Definește culoarea pentru colorarea figurilor.

Parametrul *hasura* are următoarele valori:

Constantă simbolică	Valoare
EMPTY_FILL	0
SOLID_FILL	1
LINE_FILL	2
LTSLASH_FILL	3
SLASH_FILL	4
BKSLASH_FILL	5
LTBKSLASH_FILL	6
HATCH_FILL	7
XHATCH_FILL	8
INTERLEAVE_FILL	9
WIDE_DOT_FILL	10
CLOSE_DOT_FILL	11
USER_FILL	12

Valoarea EMPTY\_FILL colorează figura folosind culoarea de fond.

Valoarea SOLID\_FILL realizează colorarea uniformă a figurii (toți pixelii din interiorul figurii au aceeași culoare).

Valorile de la LINE\_FILL(2), până la CLOSE\_DOT\_FILL(11) definesc diferite hașuri standard.

Valoarea USER\_FILL se utilizează când hașura se definește de către utilizator. Pentru a defini o hașură nestandard se utilizează funcția *setfillpattern*. Aceasta are prototipul:

```
void far setfillpattern(char far *h_utilizator, int culoare);
```

unde:

- h\_utilizator* - Este un pointer spre o zonă de memorie în care se definește, pe 8 octeți, hașura.
- culoare* - Definește culoarea de hașurare.

Șablonul se definește printr-un șir de 8 octeți. Fiecare octet definește 8 pixeli, deci un șablon definește 64 pixeli.

Modul curent utilizat la colorare se poate determina cu ajutorul funcției

*getfillsettings*. Aceasta are prototipul:

```
void far getfillsettings(struct fillsettingstype far *stilcolorare);
```

unde:

*fillsettingstype* - Este definit în fișierul *graphics.h* astfel:

```
struct fillsettingstype {  
    int pattern;  
    int color;  
};
```

unde:

*pattern* - Este componenta care definește modul de colorare utilizat la colorare (întreg din intervalul [0,12]).

*color* - Definește culoarea utilizată la colorare.

Funcția *getfillpattern* determină șablonul definit de utilizator cu ajutorul funcției *setfillpattern*. Ea are prototipul:

```
void far getfillpattern(char far *sablon);
```

La apelul funcției *getfillpattern* se atribuie, la 8 octeți din zona spre care pointează pointerul *sablon*, valorile care stabilesc hașura definită de utilizator prin apelul prealabil al funcției *setfillpattern*.

Utilizatorul poate trasa și alte figuri decât cele pentru care au fost prevăzute funcții standard de felul celor indicate mai sus. În acest scop se pot folosi funcțiile *putpixel*, *line*, *lineto*, *lnxerl*, *arc* etc., pentru a trasa elemente componente ale figurii de trasat.

Pentru a colora un domeniu închis obținut în acest fel, se folosește funcția *floodfill*. Aceasta are prototipul:

```
void far floodfill(int x, int y, int culoare_froniera);
```

unde:

*(x,y)* - Sînt coordonatele unui punct interior figurii care se colorează.

*culoare\_froniera* - Definește culoarea utilizată la trasarea conturului figurii.

Interiorul figurii se colorează în conformitate cu parametrii curenți setați prin apelul funcției *setfillstyle*.

La trasarea figurilor trebuie să se țină seama de faptul că pixelii nu sînt puncte ideale. Ei au forme dreptunghiulare. Din această cauză, figurile afișate pe ecran vor avea un aspect deformat față de forma lor ideală. Pentru

a evita acest lucru se utilizează niște factori de corecție numiți coeficienți de *aspect*. Aceștia sînt doi: unul relativ la abscisă și unul relativ la ordonată. Valorile lor se pot seta cu ajutorul funcției *setaspectratio* de prototip:

```
void far setaspectratio(int xaspect, int yaspect);
```

Valorile acestor coeficienți depind de adaptorul grafic.

Raportul (*double*) *xaspect/yaspect* se folosește adesea la trasarea curbelor definite analitic.

Valorile curente ale acestor coeficienți pot fi găsite apelînd funcția *getaspectratio* de prototip:

```
void far getaspectratio(int far *xaspect, int far *yaspect);
```

După apel, cei doi coeficienți de aspect curenți se află memorări în zonele spre care pointează parametrii *xaspect* și *yaspect*.

**Exerciții:**

19.10 Să se scrie un program care trasează linii orizontale folosind cele 4 stiluri standard și ambele grosimi.

## PROGRAMUL BXIX10

```
#include <graphics.h>
#include <conio.h>
```

```
main() /* traseaza linii orizontale folosind cele 4 stiluri standard si
        ambele grosimi */
```

```
{
    char *stil[]={
        "SOLID_LINE = 0",
        "DOTTED_LINE =1",
        "CENTER_LINE =2",
        "DASHED_LINE =3"
    };
    char *grosime[]={
        "NORM_WIDTH =1",
        "THICK_WIDTH =3"
    };
    int gd=DETECT, gm;
    int i, j;
    int x, y;

    initgraph(&gd, &gm, "c:\\borlandc\\bgi");
```

```

x=getmaxx()/4;
y=getmaxy()/8;
settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
settextjustify(LEFT_TEXT,CENTER_TEXT);
for(i=SOLID_LINE;i<=DASHED_LINE;i++)
    for(j=0;j<2;j++){
        outtextxy(x,y,stil[i]);
        outtextxy(x+textwidth(stil[i])+4,
            y,grosime[j]);
        y+=textheight(grosime[j])+4;
        if(j)
            setlinestyle(i,0,3);
        else
            setlinestyle(i,0,1);
        line(x,y,3*x,y);
        y+=20;
    }
getch();
closegraph();
}

```

9.11 Să se scrie un program care trasează următoarele figuri:

- cerc;
- elipsă;
- dreptunghi;
- patrulater.

Tipul figurii se stabilește aleator. Figurile sînt trasate folosind în mod aleator culoarea de desenare.

## PROGRAMUL BXIX11

```

#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

main() /* traseaza figuri geometrice in mod aleator */
{
    int gd=DETECT,gm;
    int i;
    int xmax,ymax,x,y;
    int poligon[10];

```

```

initgraph(&gd,&gm,"c:\\borlandc\\bgi");
outtextxy(0,0,"Pentru a termina actionati tasta\
zero");
setviewport(0,8,getmaxx(),getmaxy(),1);

srand(1993); /* seteaza sãminta sirului de numere
              pseudo-aleatoare */
xmax=getmaxx();
ymax=getmaxy();
for (;;) {
    setcolor(random(15)+1);
    switch(random(4)) {
        case 0: /* cerc */
            x=random(xmax/2)+xmax/4;
            y=random(ymax/2)+ymax/4;
            circle(x,y,random(80)+1);
            break;
        case 1: /* elipsa */
            x=random(xmax/2)+xmax/8;
            y=random(ymax/2)+ymax/8;
            ellipse(x,y,0,359,random(80)+1,
                random(60)+1);
            break;
        case 2: /* dreptunghi */
            x=random(xmax/2)+xmax/10;
            y=random(ymax/2)+ymax/10;
            rectangle(x,y,x+random(xmax/4)+1,
                y+random(ymax/4)+1);
            break;
        case 3: /* patrulater */
            for(i=0;i<8; i+=2){
                poligon[i]=random(xmax);
                poligon[i+1]=random(ymax);
            }
            poligon[8]=poligon[0];
            poligon[9]=poligon[1];
            drawpoly(5,poligon);
            break;
    } /* sfirsit switch */
    if(getch()=='0')
        break;
}
closegraph();

```

}

19.12 Să se scrie un program care colorează figurile geometrice trasate în programul precedent.

## PROGRAMUL BXIX12

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

main() /* trasaza si coloreaza figuri geometrice in mod aleator */
{
    int gd=DETECT,gm;
    int i;
    int xmax,ymax,x,y;
    int poligon[8];

    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    outtextxy(0,0,"Pentru a termina actionati tasta\
        zero");
    setviewport(0,8,getmaxx(),getmaxy(),1);

    srand(1993); /* seteaza sãmînta sirului de numere
        pseudo-aleatoare */
    xmax=getmaxx();
    ymax=getmaxy();
    setcolor(getmaxcolor());
    for (;;){
        setfillstyle(SOLID_FILL,random(15)+1);
        switch(random(4)){
            case 0: /* cerc */
                x=random(xmax/2)+xmax/4;
                y=random(ymax/2)+ymax/4;
                pieslice(x,y,0,360,random(80)+1);
                break;
            case 1: /* elipsa */
                x=random(xmax/2)+xmax/8;
                y=random(ymax/2)+ymax/8;
                fillellipse(x,y,random(80)+1,
                    random(60)+1);
                break;
            case 2: /* dreptunghi */
```

```

        x=random(xmax/2)+xmax/10;
        y=random(ymax/2)+ymax/10;
        bar(x,y,x+random(xmax/4)+1,
            y+random(ymax/4)+1);
        break;
    case 3: /* patrulater */
        for(i=0;i<8; i+=2){
            poligon[i]=random(xmax);
            poligon[i+1]=random(ymax);
        }
        fillpoly(4,poligon);
        break;
    } /* sfirsit switch */
    if(getch()=='0')
        break;
}
closegraph();
}

```

19.13 Să se scrie un program care afișează 15 dreptunghiuri colorate, pe trei rânduri, folosind toate cele 15 culori ale paletei diferite de culoarea de fond.

Sub fiecare dreptunghi se listează indicele culorii dreptunghiului, în tabloul care definește paleta curentă de culori.

După afișarea celor 15 dreptunghiuri se poate regla monitorul așa încît fiecare dreptunghi să fie vizibil.

Acest program poate fi utilizat ori de cîte ori se constată că monitorul este dereglat.

### PROGRAMUL BXIX13

```

#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main() /* afiseaza 15 dreptunghiuri colorate */
{
    int Adaptorgrafic,Modgrafic;
    int lat,inalt,x,y,i,j,culoare;
    char asculoare[5];

```



```

Adaptorgrafic = DETECT;
initgraph(&Adaptorgrafic,&Modgrafic,
        "c:\\borlandc\\bgi");
culoare = 1;
lat = 78;
inalt = 64;
x = lat/2;
y = inalt/2;
for(j = 0; j < 3; j++) {
    for( i = 0 ; i < 5 ; i++) {
        setfillstyle(SOLID_FILL,culoare);
        setcolor(culoare);
        bar(x,y,x+lat,y+inalt);
        itoa(culoare,asculoare,10);
        outtextxy(x+lat/2,y+inalt+4,asculoare);
        ++culoare;
        x += (lat/2)*3;
    }
    y += (inalt/2)*3;
    x = lat/2;
}
getch();
closegraph();
}

```

19.14 Să se scrie un program care afișează dreptunghiuri utilizând toate hașurile standard.

#### PROGRAMUL BXIX14

```

#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main() /* afiseaza dreptunghiuri cu toate hasurile standard */
{
    int Adaptorgrafic,Modgrafic;
    int lat,inalt,x,y,i,j,hasura;
    char ahas[5];

    Adaptorgrafic = DETECT;
    initgraph(&Adaptorgrafic,&Modgrafic,

```

```

        "c:\\borlandc\\bgi");
lat = 78;
inalt = 64;
x = lat/2;
y = inal/2;
for(j = 0,hasura=EMPTY_FILL; j <3; j++ ){
    for( i = 0 ; i < 5 ; i++ ){
        if(hasura >=12)
            break; /* nu mai sînt hasuri */
        setfillstyle(hasura,WHITE);
        bar(x,y,x+lat,y+inal);
        rectangle(x,y,x+lat,y+inal);
        itoa(hasura,ashes,10);
        outtextxy(x+lat/2,y+inal+4,ashes);
        ++hasura;
        x += (lat/2)*3;
    }
    if(hasura >= 12)
        break;
    y += (inal/2)*3;
    x = lat/2;
}
getch();
closegraph();
}

```

19.15 Să se scrie un program care trasează un cerc folosind ecuațiile parametrice ale cercului:

$$(1) \quad x = r \cdot \cos(g) + x_{\text{centru}};$$

$$y = r \cdot \sin(g) + y_{\text{centru}};$$

unde:

$(x_{\text{centru}}, y_{\text{centru}})$  - Sînt coordonatele centrului cercului.

$r$  - Este raza cercului.

$g$  - Este unghiul în radiani dintre axa  $Ox$  și raza  $OM$ ,  $O$  fiind originea axelor de coordonate, iar  $M$  este punctul de pe cerc de coordonate  $(x,y)$ .

Cînd unghiul  $g$  variază de la  $0$  la  $2 \cdot \pi$ , punctul  $M(x,y)$  descrie cercul de rază  $r$  și cu centru în punctul de coordonate  $(x_{\text{centru}}, y_{\text{centru}})$ .

Programul afișează două curbe pentru a pune în evidență importanța coeficienților de aspect. Prima curbă se afișează folosind ecuațiile (1), iar

cea de a doua curbă se trasează corectînd ordonata conform relațiilor de mai jos:

$$(2) \quad \begin{aligned} x &= r \cdot \cos(g) + x_{\text{centru}}; \\ y &= r \cdot (x_{\text{aspect}}/y_{\text{aspect}}) \cdot \sin(g) + y_{\text{centru}}; \end{aligned}$$

unde:

*xaspect* și *yaspect* - Sînt coeficienții de aspect obținuți prin apelul funcției *getaspectratio*.

## PROGRAMUL BXIX15

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

#define PI 3.14159265

main()
{
    int gd=DETECT,gm;
    int xaspect,yaspect;
    double FactorAspect,draza,rad;
    int x,y,g;
    int xcentru,ycentru;

    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    getaspectratio(&xaspect,&yaspect);
    FactorAspect=(double)xaspect/yaspect;
    draza=150;
    FactorAspect *=draza;
    xcentru=getmaxx()/2;
    ycentru=getmaxy()/2;

    /* traseaza un cerc fara a tine seama de coeficientii de aspect */
    for(g=0;g<360;g++){
        rad=g*PI/180.0;
        x=draza*cos(rad)+xcentru;
        y=draza*sin(rad)+ycentru;
        putpixel(x,y,WHITE);
    }
```

```

}

getch();

/* traseaza un cerc tinind seama de coeficientii de aspect */
for(g=0;g<360;g++){
    rad=g*PI/180.0;
    x=draza*cos(rad)+xcentru;
    y=FactorAspect*sin(rad)+ycentru;
    putpixel(x,y,LIGHTBLUE);
}
getch();
closegraph();
}

```

- 19.16 Să se scrie un program care deplasează o imagine pe ecran. Acest program a fost propus și realizat de Negrescu Dan și Filimon Ovidiu. Programul a apărut și în cărțile [13] și [21].



```

#include <graphics.h>
#include <conio.h>
#include <alloc.h>
#include <dos.h>

main() /* deplaseaza un vapor pe ecran */
{
    void desen_vapor() ;
    int  grmode,grdrive ;
    void *ptr ;
    void *vapor1,*vapor2 ;
    int  linia,coloana ;

    grdrive=DETECT ;
    initgraph(&grdrive,&grmode,"c:\\borlandc\\bgi") ;
    grmode=EGAH1 ;
    setgraphmode(grmode) ;
    setbkcolor(BLACK) ;
    cleardevice() ;
    desen_vapor() ;
    setfillstyle(SOLID_FILL,LIGHTRED) ;
    bar(10,0,25,10) ;
}

```

```

vapor1 = malloc(imagesize(0,0,50,50)) ;
getimage(0,0,50,50,vapor1) ;
cleardevice() ;
desen_vapor() ;
setfillstyle(SOLID_FILL,YELLOW) ;
bar(25,0,40,10) ;
vapor2 = malloc(imagesize(0,0,50,50)) ;
getimage(0,0,50,50,vapor2) ;
setbkcolor(LIGHTBLUE) ;
cleardevice() ;
linia=coloana=0;
for(;;){
    putimage(coloana,linia,vapor1,XOR_PUT) ;
    delay(100) ;
    putimage(coloana,linia,vapor1,XOR_PUT) ;
    coloana+=25 ;
    if(coloana<570){
        putimage(coloana,linia,vapor2,XOR_PUT) ;
        delay(100) ;
        putimage(coloana,linia,vapor2,XOR_PUT) ;
        coloana +=25 ;
        if(coloana>570)
            coloana=0 ;
    }
    else{
        linia+=60 ;
        coloana=0 ;
        if(linia>300)
            linia=0 ;
    }
    if(kbhit())
        break ;
}
getch() ;
closegraph() ;
free(vapor1) ;
free(vapor2) ;
}

void desen_vapor() /* deseneaza un vapor */
{
    int puncte[]={
        25,10,

```

```

        10,30,
        0,30,
        10,50,
        40,50,
        50,30,
        40,30,
        25,10
    } ;

    setcolor(CYAN) ;
    drawpoly(8,puncte) ;
    line(10,30,40,30) ;
    setfillstyle(SOLID_FILL,LIGHTMAGENTA) ;
    floodfill(25,45,CYAN) ;
    setfillstyle(SOLID_FILL,WHITE) ;
    floodfill(25,20,CYAN) ;
    setcolor(BROWN) ;
    line(25,10,25,30) ;
}

```

# BIBLIOGRAFIE

1. **C.Bohn, G.Jacopini**  
FlowDiagrams, Turning Machines and Languages with OnlyTwo  
Formation Rules  
Comm ACM, 9, 5, pag. 366 - 371, 1966
2. **Brian W. Kernighan, Dennis M. Ritchie**  
The C Programming Language  
Prentice-Hall, Inc, Englewood Cliffs  
New Jersey 1978
3. **Donald E. Knuth**  
Tratat de programarea calculatoarelor, vol. 1 - 3  
Editura Tehnică  
Bucureşti 1974
4. **Valentin Cristea şi alţii**  
Dicţionar de informatică  
Editura ştiinţifică şi enciclopedică  
Bucureşti 1981
5. **Narlan Gehani**  
C: An Advanced Introduction  
AT&T Bell Laboratories Murray Hill  
New Jersey 1985
6. **Joel E. Richardson, Michel J. Carey, Daniel T. Schuh**  
The Design of the E Programming Language  
Computer Sciences Department  
University of Wisconsin Madison

7.   \*\*\*  
BORLAND INTERNATIONAL TURBO C++:  
Getting Started  
1990
8.   \*\*\*  
BORLAND INTERNATIONAL TURBO C++:  
Programmer's Guide  
1990
9.   **Frederic Lung**  
Le Langage C++  
CNIT Paris - La Defense  
1990
10. **Setrag Khoshafian, Razmik Abnous**  
Object Orientation Concepts, Languages, Databases, User Interfaces  
John Wiley&Sons, Inc. New York  
Chichester Brisbane Toronto  
Singapore 1990
11. **Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein**  
Data Structures Using C  
Prentice - Hall International, Inc.  
1990
12. **Clara Ionescu, Ioan Zsako**  
Structuri arborescente cu aplicațiile lor  
Editura Tehnică



**București 1990**

13. **Liviu Negrescu**  
Introducere în limbajul C vol. 1 - 2  
Colecția GLOB  
1990
14. **Stroustrup Bjarne**  
The C++ Programming Language  
Second Edition  
Copyright 1991 by AT&T  
Bell Telephone Laboratories, Incorporated
15. **Liviu Negrescu**  
Limbajul C - Culegere de programe  
Fasciculele 1 - 2  
Cluj-Napoca 1991
16. **Liviu Negrescu**  
Limbajul TURBO C  
Editura LIBRIS  
Cluj-Napoca 1992
17. **Liviu Negrescu**  
Introducere în mediul de dezvoltare integrat TURBO C  
Editura LIBRIS  
Cluj-Napoca 1992
18. **Ionuț Mușlea**  
Inițiere în C++ - Programare orientată pe obiecte  
Editura MicroInformatica

**Cluj-Napoca 1992**

19. **Donald L. Shell**  
CACM 2(iulie), pag. 30-32  
1959
20. **C. A. R. Hoare**  
Quicksort  
Comp. j., 5, No. 1  
1962
21. **Liviu Negrescu**  
Introducere în limbajul C  
Editura MicroInformatica  
Cluj-Napoca 1993



str. Observatorului nr.1, bl.OS1  
3400 Cluj-Napoca  
of. PTTR Cluj-Napoca, CP 186  
tel / fax 064.198263, 123125

Cartea se adresează unui cerc larg de cititori care doresc să se inițieze în programarea și utilizarea limbajelor C și C++.

Primele două volume realizează o introducere elementară în aceste limbaje.

Volumul trei conține programe diverse în C și C++ cu aplicații la rezolvarea problemelor științifice și tehnico-ingenerești, în prelucrări de date, precum și în scrierea programelor de sistem.



I.S.B.N. 973-96980-9-3

14950 lei